VŠB – Technická univerzita Ostrava
Fakulta strojní
Katedra automatizační techniky a řízení

Návrh interpretu příkazů ve skriptech

Design of Software for Interpretation Script Statements

Student: Lukáš Plch
Vedoucí diplomové práce: Prof. Ing. Jiří Tůma, CSc.
Ostrava 2009

Prohlášení diplomanta

Prohlašuji, že jsem celou diplomovou práci včetně příloh vypracoval samostatně pod vedením vedoucího diplomové práce a uvedl jsem všechny použité podklady a literaturu.

V Ostravě …….

……..

Lukáš Plch

Prohlašuji, že

- byl jsem seznámen s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. – autorský zákon, zejména §35 – užití díla v rámci občanských a náboženských obřadů, v rámci školních představení a užití díla školního a §60 – školní dílo.
- beru na vědomí, že Vysoká škola báňská – Technická univerzita Ostrava (dále jen VŠB-TUO) má právo nevýdělečně ke své vnitřní potřebě diplomovou práci užít (§35 odst. 3).
- souhlasím s tím, že jeden výtisk diplomové práce bude uložen v Ústřední knihovně VŠB-TUO k prezenčnímu nahlédnutí a jeden výtisk bude uložen u vedoucího diplomové práce. Souhlasím s tím, že údaje o diplomové práci, obsažené v Záznamu o závěrečné práci, umístěném v příloze mé diplomové práce, budou zveřejněny v informačním systému VŠB-TUO.
- bylo sjednáno, že s VŠB-TUO, v případě zájmu z její strany, uzavřu licenční smlouvu s oprávněním užít dílo v rozsahu §12 odst. 4 autorského zákona.
- bylo sjednáno, že užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití mohu jen se souhlasem VŠB-TUO, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly VŠB-TUO na vytvoření díla vynaloženy (až do jejich skutečné výše).

V Ostravě ……

……..

Lukáš Plch

Lukáš Plch
Ciolkovského 462
Hlubočky Mariánské Údolí
783 65

ANOTACE DIPLOMOVÉ PRÁCE

PLCH, L.: *Návrh interpretu příkazů ve skriptech*. Ostrava: katedra automatizační techniky a řízení, Fakulta strojní VŠB-TU Ostrava, 2009. 53 stran. Diplomová práce, vedoucí: Prof. Ing. Jiří Tůma, CSc.

Tato práce se zabývá návrhem skriptovacího jazyka, který dokáže matematicky zpracovávat vzorky signálu. V úvodních kapitolách je popsána teorie potřebná k pochopení problematiky diplomové práce. Následuje popis známých skriptovacích jazyků (např. M-file pro MATLAB, Visual Basic – Scripting Edition) a návrh svého skriptovacího jazyka, který umí zápis příkazů pro výpočet vzorců nebo celých algoritmů obsahujících podmíněné příkazy a cykly. Dalším krokem je navrhnout software pro rozklad libovolného výrazu a posloupností příkazů užitím objektového programování. Podstatou diplomové práce je vytvoření komponenty, která demonstruje vlastnosti vytvořeného skriptovacího jazyka v reálné aplikaci. V závěru této práce je zhodnocení dosažených výsledků a návrh směrů dalšího řešení.

ANNOTATION OF THE THESIS

PLCH, L.: *Design of Software for Interpretation Script Statements*. Ostrava: Department of Control Systems and Instrumentation, Faculty of Mechanical Engineering VŠB – Technical University of Ostrava, 2009. 53 pages.
Thesis, head: Prof. Ing. Jiří Tůma, CSc.

This thesis deals with the concept of a scripting language which is intended for processing signal samples. There is a theory needed to understand the dillemas of diploma thesis described in initial chapters. Here is a description of well-known scripting language (e.g. M-file for MATLAB, Visual Basic - Scripting Edition) and a proposal for its scripting language that can write commands for calculation formulas, or algorithms, containing a batch of commands and loops. The next step is to design software for decomposition expressions and a sequence of commands using the object oriented programming. Software is based on component creation, which demonstrates the characteristics of scripting language created in a real application. In conclusion, this work is to evaluate the achievements and the proposal of further solutions.

Obsah

1 Úvod

Hlavním cílem této práce je vytvořit skriptovací jazyk, který dokáže matematicky zpracovávat naměřené vzorky signálu v reálné aplikaci. Smyslem skriptovacího jazyka je poskytnout uživateli možnost doprogramovat vlastní algoritmy. Skriptování nabízí výhody, pomocí nichž můžeme docílit efektivní a automatizované práce se signály (vektory a maticemi).

V úvodu mé práce se nejprve seznámím s teorií překladačů a principy konstrukce překladačů, které jsou potřebné pro pochopení problematiky zadané práce.

Prvním bodem zadání diplomové práce je popsat známé skriptovací jazyky, (např. M-file pro MATLAB, Visual Basic – Scripting Edition) a na základě těchto vzorů navrhnout svůj skriptovací jazyk, který má umět zápis příkazů pro výpočet vzorců nebo celých algoritmů obsahujících podmíněné příkazy a cykly.

Následuje popis ActiveX prvku Microsoft Script Control, jeho metod a vlastností.

Dalším dílčím úkolem zadání diplomové práce je navrhnout software pro rozklad libovolného výrazu a posloupností příkazů užitím objektového programování.

Předposlední úloha zadání spočívá v začlenění mého software do reálné aplikace tak, aby demonstrovala vlastnosti vytvořeného skriptovacího jazyka.

Na závěr diplomové práce mám zhodnotit dosažené výsledky a navrhnout směry dalšího řešení.

2 Základy teorie překladačů

2.1 Gramatika

Gramatika jazyka je tvořena množinou všech gramatických pravidel. Gramatickými pravidly je určena pouze *syntaxe jazyka*, to znamená, že je určena přípustná struktura vět jazyka a prvotní jednotky, které je možné na daném místě použít. Pokud libovolná věta splňuje podmínky definované gramatikou, patří do daného jazyka.

Význam vět určuje *sémantika*. Syntaxe a sémantika navzájem spolu souvisí. Syntaxe definuje strukturu věty, která je zase základem pro určení jejího významu. Syntaktická správnost věty však ještě nezaručuje její sémantickou správnost.

2.1.1 Základní gramatika

Analýza formálního jazyka a gramatiky vyžaduje specifikaci objektů, se kterými budeme pracovat. Jedná se o prvotní symboly nazývané *terminální symboly* nebo *terminály*, jednotky specifikované pomocí jiných jednotek *neterminální symboly* nebo *neterminály*, gramatická pravidla nazývané *přepisovací pravidla*.

Gramatika generující formální jazyk je čtveřice:

$$G = (N, T, P, S)$$

kde

- N je konečná množina neterminálních symbolů
- T je konečná množina terminálních symbolů

  $T \cap N = \emptyset$
- P je konečná množina přepisovacích pravidel
- S je startovací (počáteční) symbol gramatiky

Přepisovací pravidla jsou tvaru:

$$A \rightarrow X$$

kde

- A je neterminál
- X je řetězec symbolů: $X \in (N \cup T)^*$

2.1.2 Rozšířená gramatika

Rozšířená gramatika generující formální jazyk je čtveřice:

$$G' = (N \cup \{S'\},\ T,\ P \cup \{S' \rightarrow S\},\ S')$$

Rozšířená gramatika G' je ekvivalentní s gramatikou G.

2.1.3 Překladová gramatika

Překladová gramatika je pětice:

$$G_P = (N,\ T,\ D,\ P,\ S)$$

kde

- N je konečná množina neterminálních symbolů
- T je konečná množina terminálních symbolů

  $T \cap N = \emptyset$
- D je konečná množina výstupních symbolů

  $T \cap D = \emptyset \quad \wedge \quad (T \cup D) \cap N = \emptyset$
- P je konečná množina pravidel gramatiky
- S je startovací (počáteční) symbol gramatiky

2.2 LR gramatiky jazyků

LR gramatiky reprezentují největší třídu gramatik, ke kterým lze vždy sestrojit deterministický syntaktický LR analyzátor směrem zdola nahoru. Tyto analyzátory se nazývají LR, protože čtou vstupní řetězec zleva (Left) a vytvářejí pravý (Right) rozklad.

Důležitost LR jazyků plyne z těchto dvou skutečností:

1. Pro všechny programovací jazyky, které lze popsat bezkontextovou gramatikou můžeme prakticky sestrojit syntaktické LR analyzátory.
2. Metody syntaktické analýzy LR jazyků jsou obecnější, než metody analýzy jednoduchých nebo operátorových precedenčních jazyků nebo metody deterministické syntaktické analýzy shora dolů, a přitom vedou ke stejně efektivním analyzátorům.

Modelem syntaktického analyzátoru pracujícího metodou zdola nahoru je zásobníkový automat. Tento zásobníkový automat je obecně nedeterministický, nelze jej přímo použít jako syntaktický analyzátor, a proto jej musíme vhodně upravit [BABIUCH M. 1998].

Podobně jako při analýze shora dolů můžeme i v tomto případě využít následující informace:

- Informace o dosud nepřečtené části vstupního řetězce,
- Informace o dosavadním průběhu analýzy.

2.2.1 Silné LR gramatiky

Gramatiky, pro které stačí při analýze metodou zdola nahoru k deterministickému rozhodování pouze informace o jednom symbolu na vrcholu zásobníku a o nejbližších *k* symbolech z dosud nepřečtené části vstupního řetězce, nazýváme silné LR gramatiky:

- Silné LR(0) gramatiky, nevyužívají informaci o vstupu,
- Silné LR(k) gramatiky, využívající informaci o vstupu (k) symbolů.

2.2.2 Slabé LR gramatiky

Pro slabé LR gramatiky je při deterministické syntaktické analýze již nutno využívat informaci o dosavadním průběhu analýzy. Je tedy zřejmé, že slabé LR gramatiky zahrnují i silné LR gramatiky.

Kromě slabých LR(k) gramatik se zabýváme i jejich podtřídami:

- LR(0) gramatiky,
- SLR(k) gramatiky (Simple LR),
- LALR(k) gramatiky (Look Ahead LR).

Důvody pro zavedení těchto podtříd jsou tyto:

- Konstrukce syntaktického analyzátoru pro tyto podtřídy je jednodušší než pro LR(k) gramatiky,
- tabulky pro syntaktický analyzátor mají menší rozsah,
- v některých případech je syntaktická analýza rychlejší.

Pro obě třídy (silné i slabé) se používá až na drobné modifikace tentýž algoritmus syntaktické analýzy, založený na principu zásobníkového automatu. Pro rozhodování o tom, zda a jakou provést v daném okamžiku redukci se ve všech případech používá rozkladové tabulky, ve které jsou uvedeny všechny potřebné informace. Tato tabulka se sestrojuje na základě bezkontextové gramatiky [BABIUCH M. 1998].

2.3 Atributovaný překlad

2.3.1 Základní pojmy

Základními pojmy budou pojmy *atribut, atributovaný symbol, atributovaný řetězec*.

Atributem budeme rozumět veličinu, která může nabývat hodnot z nějaké množiny. Tato množina je oborem hodnot atributu. Atribut můžeme přirovnat k proměnné v programu, která má definován typ.

Atributovaný symbol je symbol nějaké abecedy, ke kterému je přiřazena konečná množina atributů. Množina atributů přiřazených symbolu může být prázdná.

Atributovaný řetězec nad abecedou *A* je řetězec atributovaných symbolů z A. V dalším výkladu budeme používat tato označení: Atribut *a* přiřazený symbolu *x* označíme *x.a*. Atributovaný symbol *x* s přiřazenými atributy a_1, a_2, …, a_n označíme *x* [$x.a_1$, $x.a_2$, …,$x.a_n$], kde $x.a_i$ mohou být nahrazeny hodnotami z oboru hodnot atributu a_i, $1 \leq i \leq n$ [BABIUCH M. 1998].

2.3.2 Atributovaný překlad – definice

je relace

$$Z_A \subset T^* \times D^*$$

kde

*T** je množina vstupních atributovaných řetězců

*D** je množina výstupních atributovaných řetězců

2.3.3 Příklady atributovaného překladu

Př. 1:

Je dán atributovaný překlad takto[1]: Vstupními atributovanými řetězci jsou výrazy nad abecedou {*a*, +, *, (,)}. Symbolu *a* je přiřazen atribut *x*, jehož obor hodnot je množina celých čísel. Množiny atributů přiřazené symbolům +, *, (,) jsou prázdné. Výstupní abeceda je tvořena jediným symbolem *v*, který má přiřazen atribut *y* s množinou celých čísel jako oborem hodnot.

Předpokládejme, že atributovaný symbol *a* [*a.x*] je operand ve výrazu, kde hodnota atributu *a.x* představuje hodnotu operandu. Atributovaný symbol *v* [*v.y*] je výsledek výpočtu výrazu a *v.y* je hodnota výrazu. Do uvažovaného atributovaného překladu patří např. Tyto dvojice:

(*a* [10] ,*v* [10]),

(*a* [5] + *a* [6] ,*v* [11]),

(*a* [3] * *a* [4] + *a* [2] ,*v* [14]).

Př. 2:

Je dán atributovaný překlad z příkladu 1 s tím, že změníme význam atributu *x* symbolu *a*. Budeme předpokládat, že *a.x* bude relativní adresa, na které je uložena hodnota operandu *a*. Výpočet hodnoty výrazu můžeme pak provést za těchto předpokladů:

- je dána adresa *p*, ke které jsou vztaženy relativní hodnoty operandů,

- je dána funkce **vyber**(*x*), která z každé adresy *x* vybere hodnotu.

Jestliže *p* = 100, **vyber**(102) = 3, **vyber**(103) = 5, **vyber**(104) = 6, pak do uvažovaného překladu patří dvojice (*a* [3] * *a* [4] + *a* [2], *v* [33]).

V tomto překladu se objevil další prvek atributovaného překladu, tj. závislost atributovaného výstupního řetězce na parametru, kterým je v našem případě adresa *p*. Z uvedených příkladů je vidět, že hodnoty atributů výstupních symbolů mohou záviset:

- na hodnotách atributů vstupních symbolů,
- na struktuře vstupního řetězce,
- na zadaných parametrech.

Nyní si ukážeme způsob výpočtu atributu výstupního symbolu *v* ve výše uvedené dvojici. Dvojice (*a* * *a* + *a*,*v*) patří do formálního překladu, který je generován např. gramatikou G_{PG} :

G_{PG} = ({S, E, F}, {a, +, *, (,)}, {v}, R, S), kde R obsahuje pravidla:

$S \rightarrow E\ v$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

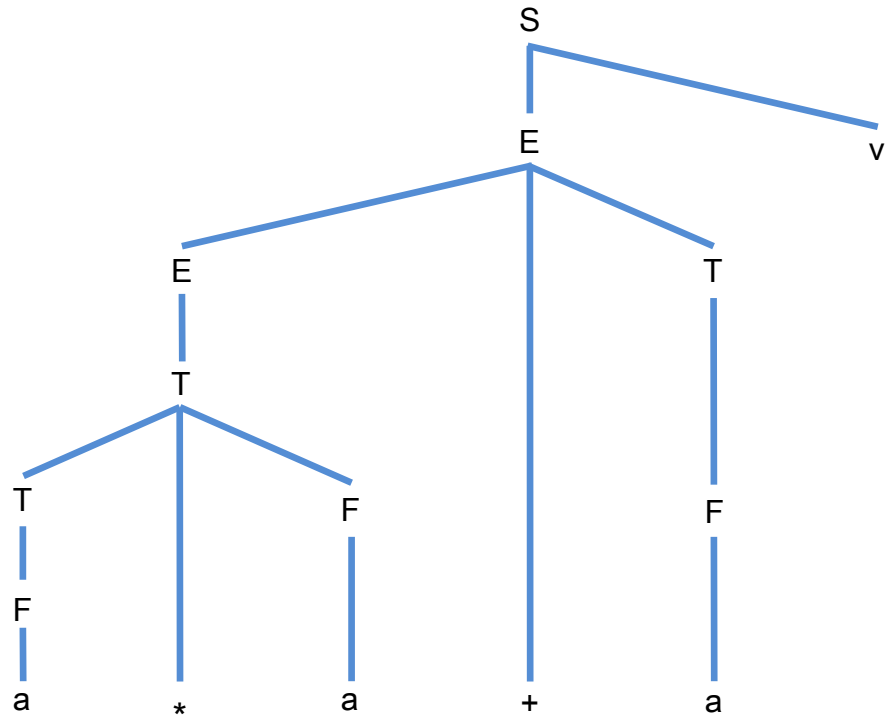
$F \rightarrow a \mid (E)$

Překladový strom pro dvojici (*a* *a* + *a, v*) má tvar uvedený na obr. 1. K tomu, abychom mohli určit hodnotu atributu v.y, je třeba určit hodnotu výrazu, který generuje symbol *E* v pravidle $S \rightarrow E\ v$. K výpočtu hodnoty tohoto výrazu potřebujeme provést součet hodnot výrazu, které generují symboly *E* a *T* v pravidle $E \rightarrow E + T$. Budeme-li pokračovat tímto způsobem, zjistíme, že je třeba provést výpočet hodnot podvýrazů, které generují všechny neterminální symboly v překladovém stromu. Přitom si všimneme, že pro různé výskyty téhož neterminálního symbolu, mohou mít jím generované podvýrazy různé hodnoty.

Z těchto důvodů je vhodné přiřadit všem neterminálním symbolům atribut, jehož hodnotou bude hodnota podvýrazu generovaného příslušným neterminálním symbolem. Označme tento atribut *h*.

Dále je třeba stanovit sémantická pravidla pro výpočet hodnot tohoto atributu u jednotlivých neterminálních symbolů. Např. pro neterminální symbol *E*

na levé straně pravidla $E \rightarrow E + T$ má sémantické pravidlo tvar: $E.h := E.h + T.h$, kde $E.h$ na levé straně sémantického pravidla je atribut symbolu E z levé strany syntaktického pravidla $E \rightarrow E + T$ a $E.h$ na pravé straně sémantického pravidla je atribut symbolu E z pravé strany téhož syntaktického pravidla [BABIUCH M. 1998].



Obr. 1: Překladový strom výrazu ($a * a + a$)

Pro neterminální symbol F se výpočet hodnoty atributu h redukuje na vyhodnocení funkce **vyber**($a.x$ + p). Při určení argumentu této funkce je třeba použít adresu p, která musí být zadána. Opět je vhodné přiřadit všem neterminálním symbolům adresu p jako atribut a stanovit pravidla pro výpočet jeho hodnoty. Zadanou hodnotu je vhodné přiřadit jako počáteční hodnotu jednomu atributu, nejlépe u kořene překladového stromu. Po provedení všech těchto úprav budou mít jednotlivé symboly gramatiky přiděleny podle tab. 1.

Tab. 1 - Atributy přidělené jednotlivým symbolům

Symboly	Atributy
S	*p*
E, T, F	*h, p*
a	*x*
v	*y*

Sémantická pravidla pro výpočet hodnot atributů přidělíme jednotlivým pravidlům překladové gramatiky. Vzhledem k tomu, že se v jednom syntaktickém pravidle může vyskytnout tentýž symbol vícekrát a může mít různé hodnoty atributů, zavedeme označení, které umožňují rozlišit jednotlivé výskyty téhož symbolu takto: Výskyt symbolu na levé straně pravidla označíme horním indexem ^{0}. Výskyty téhož symbolu na pravé straně pravidla označíme horními indexy ^{1},^{2}, atd. Toto označení je použito v následující tabulce.

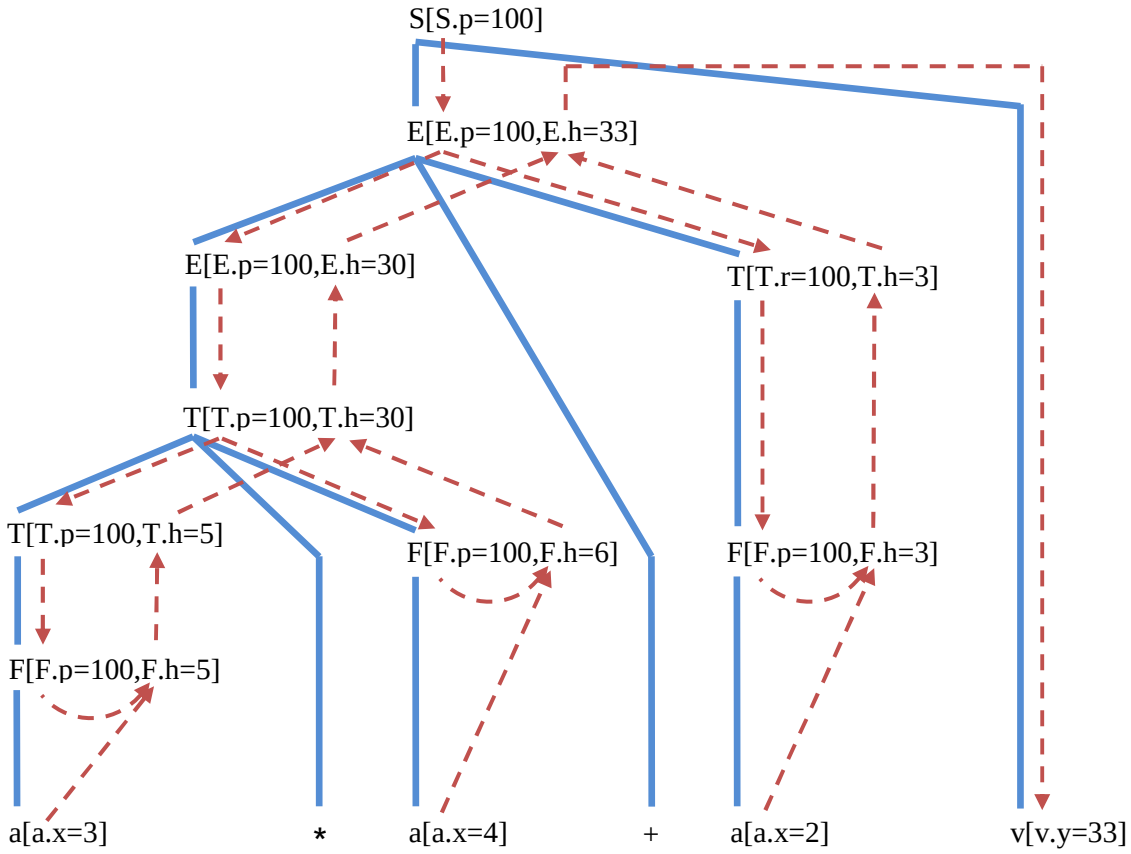
Tab. 2 – Pravidla pro výpočet atributů

Pravidla překladové gramatiky	Pravidla pro výpočet atributů
$S \rightarrow E\ v$	$E.p := S.p\ v.y := E.h$
$E \rightarrow E + T$	$E.p := E.p\ E.h := E.h + T.h$ $T.p := E.p$
$E \rightarrow T$	$T.p := E.p\ E.h := T.h$
$T \rightarrow T * F$	$E.p := T.p\ T.h := T.h + F.h$ $F.p := T.p$
$T \rightarrow F$	$F.p := T.p\ T.h := F.h$
$F \rightarrow a$	$F.h :=$ vyber$(a.x + F.p)$
$F \rightarrow (E)$	$E.p := F.p\ F.h := E.h$

Překladový strom na obr.1 můžeme doplnit tak, že ke každému symbolu připojíme hodnoty jeho atributů. Získáme tak *atributovaný překladový strom*. Na obr.2 je uveden atributovaný překladový strom pro vstupní větu *a* [3] ∗ *a* [4] + *a* [2] z př.1. Přitom platí *S.p* = 100, **vyber**102 = 3, **vyber**103 = 5, **vyber**104 = 6.

Čárkovanými šipkami jsou na obr.2 naznačeny závislosti mezi atributy. Mezi těmito závislostmi, které představují tok potřebné informace, jsou závislosti dvou typů. Jeden typ závislosti se týká atributu *h.* Hodnota tohoto atributu u určitého neterminálního symbolu závisí na informacích obsažených uvnitř podstromu, jehož kořenem je příslušný neterminální symbol. Atributy s tímto typem závislosti budeme nazývat *syntetizované.*

Druhý typ závislosti se týká atributu *p.* Hodnota tohoto atributu u určitého neterminálního symbolu závisí na kontextu, ve kterém se příslušný podstrom nachází. Atributy s tímto typem závislosti budeme nazývat *dědičné* [BABIUCH M. 1998].



Obr. 2 – Atributovaný překladový strom věty (*a* [3] ∗ *a* [4] + *a* [2])

2.3.4 Atributová překladová gramatika

je čtveřice

$$G_{AP} = (G_P, A, V, F)$$

kde

G_P je překladová gramatika $G_P = (N, T, D, P, S)$, která se nazývá *základní překladová gramatika* atributové překladové gramatiky G_P a kde každé pravidlo je zapsáno ve tvaru:

$$(r)\ X_0 \rightarrow X_1 X_2 \ldots X_3$$

Kde $n_r \geq 0$, $X_0 \in N$, $X_k \in (N \cup T \cup D)$ pro $1 \leq k \leq n_r$.

A je konečná množina atributů, která je rozdělena na dvě disjunktní podmnožiny, množinu syntetizovaných atributů S a dědičných atributů I. Pro každý atribut a je zadán *obor hodnot H (a)*, kterých může atribut nabývat.

V je zobrazení, které každému neterminálnímu symbolu $X \in N$ přiřazuje hodnotu atributů $a(X) \in$ A . Tato množina se dělí na dvě disjunktní podmnožiny $I(X)$ a $S(X)$. $I(X)$ je množina dědičných atributů, $S(X)$ je množina syntetizovaných atributů. Každému vstupnímu symbolu $X \in T$ je přiřazena množina syntetizovaných atributů $S(X)$ a každému výstupnímu symbolu $X \in$ je přiřazena množina dědičných atributů $I(X)$.

F je konečná množina sémantických pravidel. Pro každý symbol X_k ($1 \leq k \leq n_r$) na pravé straně pravidla $r \in R$ a pro jeho dědičný atribut d je dáno sémantické pravidlo

$$d := f_{rdk}(a_1, a_2, \ldots, a_m),$$

kde $a_1, a_2, \ldots, a_m$ jsou atributy symbolů v témže pravidle r .

Pro každý syntetizovaný atribut s symbolu X_0 na levé straně pravidla $r \in R$ je definováno sémantické pravidlo

$$s := f_{rso}(a_1, a_2, \ldots, a_m),$$

kde $a_1, a_2, \ldots, a_m$ jsou atributy symbolů v pravidle r .

2.3.5 Výpočet hodnot atributů

Problém výpočtu hodnot atributů můžeme formulovat takto: Nechť je dán atributovaný vstupní řetězec. Pro tento řetězec sestrojíme v základní gramatice překladový strom. Ke kořeni stromu doplníme zadané hodnoty dědičných atributů a k listům, které odpovídají vstupním symbolů, doplníme zadané hodnoty syntetizovaných atributů. Pak tento strom nějakým způsobem procházíme a přitom podle sémantických pravidel přiřazených jednotlivým pravidlům překladové gramatiky vypočítáváme hodnoty atributů.

Pro tento výpočet je třeba nalézt takové uspořádání atributů jednotlivých uzlů, v němž každý atribut je následovníkem těchto atributů, jejichž hodnoty jsou zapotřebí pro výpočet hodnoty daného atributu. Jsou-li v závislostech mezi atributy uzlů překladového stromu cykly, pak takové uspořádání neexistuje a atributová gramatika není dobře zadaná [BABIUCH M. 1998].

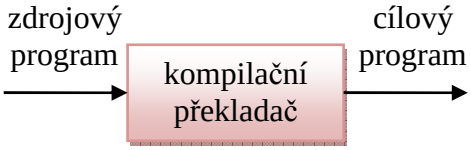
3 Principy konstrukce překladačů

3.1 Kompilační a interpretační překladače

Překladač je program, který jako vstupní data zpracovává text zapsaný v nějakém jazyce. Tomuto jazyku budeme říkat *zdrojový jazyk* a vstupnímu programu budeme říkat *zdrojový program*. V závislosti na typu zdrojového programu rozdělujeme překladače do dvou kategorií:

- kompilační překladače (assemblery a kompilátory),
- interpretační překladače (interprety).

Kompilační překladač čte zdrojový program zapsaný ve zdrojovém jazyce a překládá (transformuje) jej na ekvivalentní cílový program zapsaný v cílovém jazyce. Kompilační překladač je znázorněn na obr. 3.



Obr. 3 Kompilační překladač

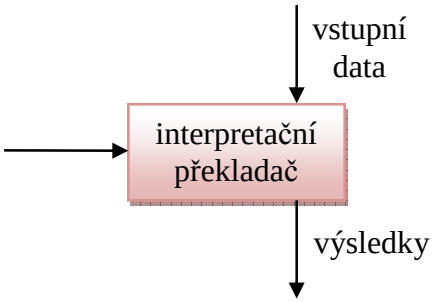
Pokud je zdrojovým jazykem jazyk symbolických instrukcí a cílovým jazykem jazyk relativních nebo absolutních adres (strojový kód), potom se příslušný kompilační překladač obvykle nazývá assembler. V případě, že zdrojovým jazykem je vyšší programovací jazyk (Pascal, C, Fortran) a cílovým jazykem je jazyk symbolických instrukcí, jazyk relativních nebo absolutních adres (strojový kód), pak se příslušný překladač obvykle nazývá kompilátor.

Při práci kompilačního překladače vzniká cílový program. Samotné provádění cílového programu (zadání vstupních dat, výpočet a získání výsledků) je činnost časově oddělena od překladu.

Interpretační překladač čte zdrojový program, analyzuje jej a zajišťuje provádění operací, které odpovídají jednotlivým příkazům zdrojového programu. Této činnosti se říká interpretace zdrojového programu. Při interpretaci příkazů

vstupu se čtou vstupní data, při interpretaci příkazů pro výstup se provádí výstup výsledků. Při práci interpretačního překladače nevzniká cílový program. Interpretační překladač je znázorněn rovněž na obr. 4.

Při rozboru existujících překladačů bychom zjistili, že mají velmi různorodou strukturu. I přes značnou rozdílnost jednotlivých překladačů je možné vysledovat jejich společné rysy [BABIUCH M. 1998].



Obr. 4 Interpretační překladač

3.2 Struktura překladače

Kompilační překladač můžeme rozdělit na čtyři hlavní části podle obr. 5. Jednotlivé části překladače představují funkční celky, které provádějí dílčí operace, na něž můžeme celý proces překladu rozdělit.

3.2.1 Lexikální analýza

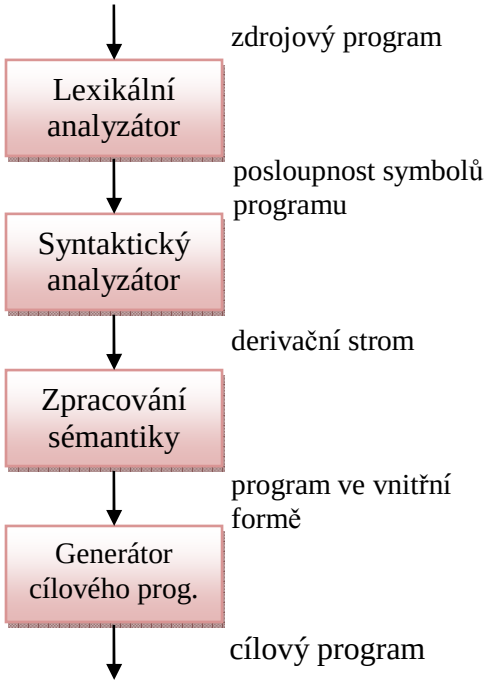
Lexikální analyzátor čte postupně znaky zdrojového programu a vytváří z nich lexikální symboly programu jako např. čísla, identifikátory, klíčová slova a jednoznakové a víceznakové omezovače. Přitom lexikální analyzátor vynechává znaky, které pro program nemají význam (např. mezery, komentáře apod.).

3.2.2 Syntaktická analýza

Výstupem lexikálního analyzátoru je posloupnost symbolů programu. Tato posloupnost symbolů je vstupem pro syntaktický analyzátor. Syntaktický analyzátor analyzuje syntaktickou strukturu programu a přitom zjišťuje, zda je program zapsán syntakticky správně. Výstupem syntaktického analyzátoru je informace o syntaktické struktuře programu nebo signalizace chyby.

3.2.3 Zpracování sémantiky

Při zpracování sémantiky se vytváří vstup pro generování cílového programu. Dále se při zpracování sémantiky kontroluje dodržení kontextových podmínek v programu. Mezi tyto kontroly patří například zjištění, zda jsou všechny použité identifikátory popsány (v Pascalu), zda souhlasí typy levé a pravé strany v přiřazovacím příkazu (ve Fortranu) apod.



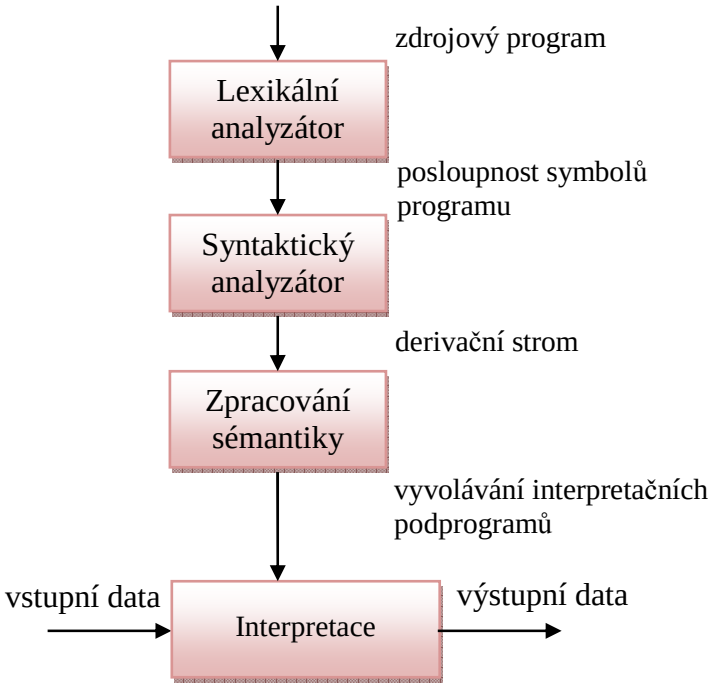
Obr. 5 Struktura kompilačního překladače

3.2.4 Generování cílového programu

Vstupem generátoru cílového programu kompilačního překladače je program ve vnitřní formě. Na základě tohoto programu jsou generovány instrukce cílového programu tak, aby vzniklý program byl ekvivalentní překládanému zdrojovému programu. V této části překladače je možno také provádět optimalizaci cílového programu, která umožňuje větší efektivnost vytvořeného programu. Výsledkem práce generátoru cílového programu je posloupnost instrukcí v jazyce symbolických instrukcí, relativních nebo absolutních adres.

Podobně jako kompilační překladač můžeme i interpretační překladač rozdělit na několik částí podle obr. 2. První tři části (lexikální analýza, syntaktická analýza a zpracování sémantiky) se neliší od stejných částí v kompilačním

překladači. Nová je pouze část, která provádí operace potřebné k interpretaci příkazů zdrojového programu. Po zpracování sémantiky je interpretační části předána informace o požadovaných operacích a interpretační část tyto operace okamžitě provádí. To znamená, že po zpracování sémantiky jsou požadované operace ihned interpretovány [BABIUCH M. 1998].



Obr. 6 Struktura interpretačního překladače

3.3 Definice programovacího jazyka

Gramatika je podrobně prostudovaný a formálně nadefinovaný nástroj pro generování jazyka. Popis programovacího jazyka se skládá z popisu *syntaxe* a *sémantiky.* Popisem syntaxe programovacího jazyka je vymezena množina všech správně vytvořených programů - řetězců jazyka. Sémantika pak stanoví význam řetězců, tvořících syntaktickou stránku jazyka.

Já jsem se na konec rozhodl po společné konzultaci s odbornými asistenty a vedoucím mé diplomové práce, že nebudu pokračovat ve vytváření nového programovacího jazyka podle těchto pravidel, ale využiji hotového ActiveX prvku Microsoft Script Control, který rozšířím o matematické a vědecké funkce potřebné pro zpracování signálů.

4 Skriptovací jazyky

Skript je zdrojový kód programu vytvořený v programovacím jazyku tzv. *skriptovacím*, který má přesně stanovenou formální gramatiku (tj. pravidla, syntaktické elementy, jazykové konstrukty atd.) a který je interpretován, tj. čten a spouštěn za běhu programu speciálním procesem, interpretem.

Typický skript těží z výhody, že se nemusí kompilovat do strojového kódu a často tvoří rozšiřitelnou (parametrickou) část nějakého softwarového projektu, která se může měnit, aniž by bylo potřeba pokaždé rekompilovat hlavní spustitelný soubor. Takto skripty najdeme u her, složitějších softwarových řešení, jako hlavní součást dynamických internetových stránek nebo v aplikacích, kde je potřeba numerických výpočtů.

Interpretované jazyky jsou pomalejší, ale nemají tak velké formální požadavky (není potřeba inicializovat proměnnou, její datový typ se může za běhu měnit, ukazatele jsou zbytečné). Hlavní nevýhodou těchto jazyků je, že se musejí vždy spouštět v interpretu. Do této skupiny patří všechny skriptovací jazyky (M-file pro MATLAB, JavaScript, VBScript, PHP, Perl, Python, Ruby, atd.).

Úroveň skriptů a jejich jazyků se může velice lišit. Některé skriptovací jazyky (např. PHP nebo Python) nabízejí třeba objektově orientované programování. Na druhou stranu, jiné soubory, které obsahují tolik konfiguračních parametrů jiného programu, mohou být též označovány jako skripty. Většinou by ale skriptovací jazyk měl být alespoň na takové úrovni, aby dovoloval pracovat s proměnnými nebo umožňoval větvení programu.

Výhody skriptovacího jazyka:

- Není nutné mít nainstalovaný kompilátor a provádět po každé změně kódu kompilaci
- Snadnější údržba, vývoj a správa kódu
- Některé skripty umožňují interpretaci kódu z řetězce (jako například funkce Eval() v PHP a jazyce VBScript). Něco takového překládané programy z principu nedovedou
- Interpret obvykle poskytuje programátorovi zabudovaný abstraktní datový typ asociativní pole

Nevýhody skriptovacího jazyka:

- Interpretace stojí určitý čas a nikdy nebude tak rychlá jako spouštění přeloženého (a optimalizovaného) programu
- trochu vyšší paměťová náročnost. Interpret musí být spuštěn, a tedy zabírá určitou operační paměť
- Skriptovací jazyky mají většinou větší omezení než překládané programovací jazyky (např. co do přístupu paměti, ovládání tzv. rukojetí procesů a kontextových zařízení apod.)

4.1 M-file pro MATLAB

MATLAB je interaktivní systém pro vědecké a technické výpočty, založený na maticovém kalkulu. Umožňuje řešit velkou oblast numerických problémů, snadnou a rychlou práci s maticemi reálných nebo komplexních čísel, aniž byste museli programovat vlastní program. Zároveň umí vypočítané výsledky graficky znázornit. Název MATLAB vznikl zkrácením z MATrix LABoratory.

4.1.1 Výrazy a proměnné ve vzorcích

MATLAB je "výrazový" jazyk. Výrazy, které zadáte, jsou přečteny a vyhodnoceny. Příkazy MATLABu mají obvykle tvar *proměnná = výraz*, nebo jen jednoduše *výraz.*

Výraz je posloupnost konstant, názvů proměnných, operátorů (včetně kulatých závorek) a volání funkcí. Pokud je výraz smysluplný (MATLABem vyhodnotitelný), tak po jeho zapsání do okna Command Window a stisku klávesy enter je ihned vyhodnocen. Vyhodnocením výrazu vzniká vždy nějaká hodnota. Výslednou hodnotu výrazu můžeme uložit do nějaké proměnné. Pokud jsou jméno proměnné a znak rovnítka „=“ vynechány, automaticky se vytvoří proměnná ans (answer = odpověď), do které se přiřadí výsledek. Pokud nepotřebujeme vypočtenou hodnotu vidět, lze její zobrazení potlačit středníkem. Obvykle se při potlačeném výpisu výsledku používá přiřazení hodnoty výrazu do nějaké proměnné, protože jinak vypočtená hodnota zanikne.

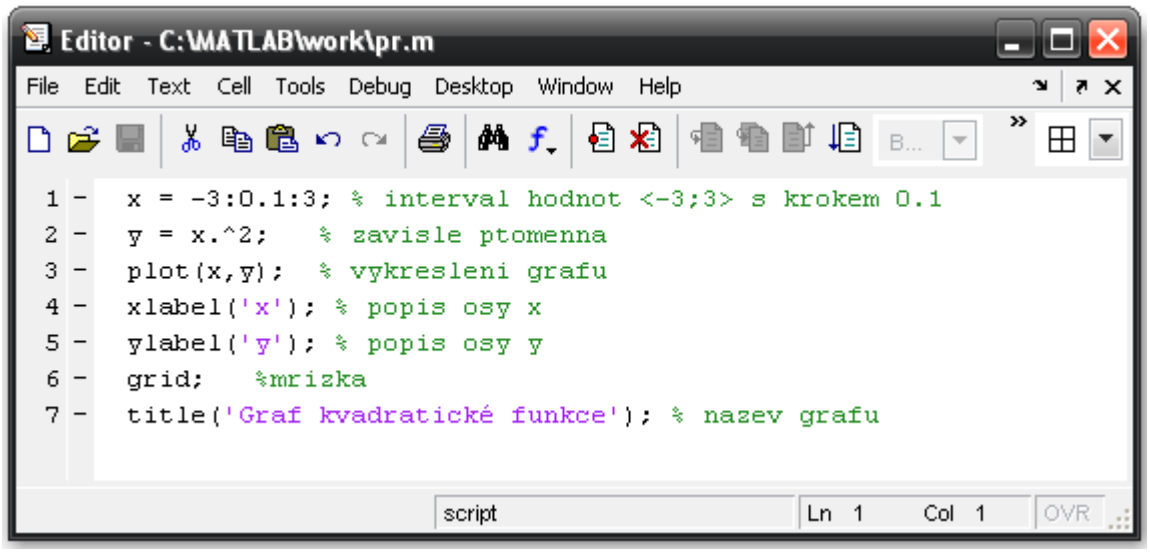
Vyhodnocením výrazu se vytvoří matice, která je zobrazena na monitor a přiřazena do proměnné pro pozdější použití. Proměnná je objekt, který má svůj

název, typ a obsah (hodnotu). Název proměnné může obsahovat až 31 znaků a to písmena anglické abecedy (a-z, A-Z), číslice (0-9) a podtržítko (_). Číslicí název začínat nesmí. V názvech jsou rozlišována velká a malá písmena (tzv. vlastnost case-sensitive). MATLAB nerozlišuje různé typy proměnných. Každá proměnná je matice a ta může obsahovat jakákoliv komplexní čísla. Z hlediska rozměrů matic rozlišujeme proměnné:

- matice (mxn, kde m>1, n>1)
- vektory (mx1 nebo 1xn)
- skaláry (1x1, tedy jen jedno číslo) kde m, n…počet řádků, sloupců

4.1.2 Vytváření Skriptů v M-file

Skript je posloupnost příkazů uložených do souboru. Každý skript pracuje s proměnnými pracovního prostředí, takže může vytvářet nové nebo mazat či měnit vybrané proměnné. Výsledky skriptu tedy zůstávají v pracovním prostředí i po jeho skončení. Skripty samozřejmě mohou volat jiné skripty nebo funkce, vytvářet grafická okna, vypisovat do Command Window, atd.



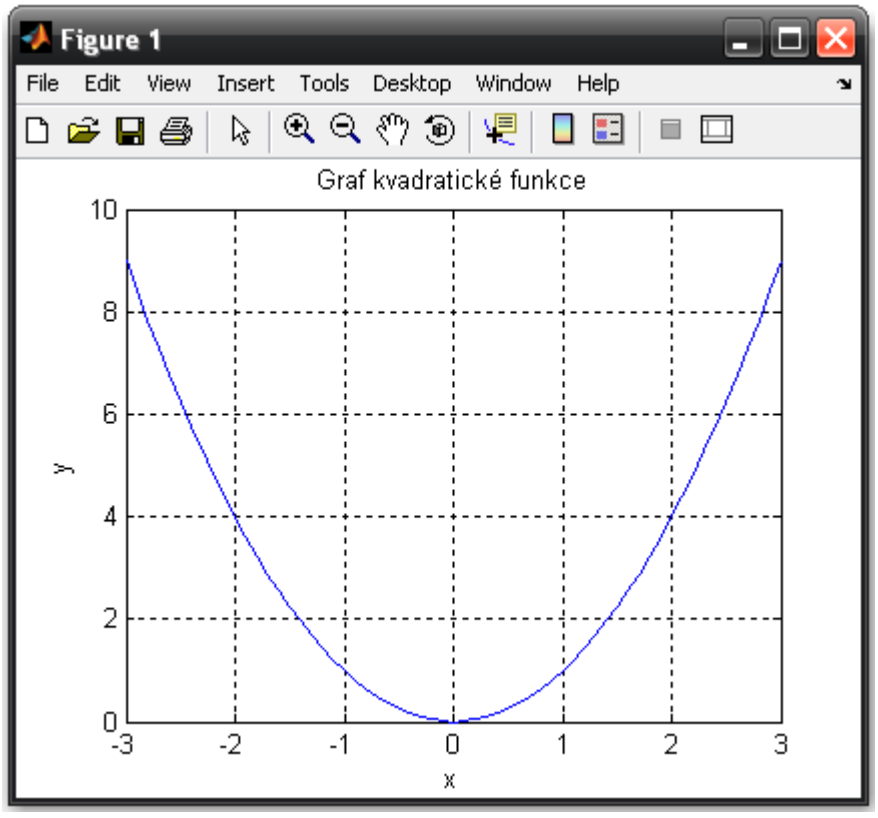
Obr. 7 Příklad skriptu v M-souboru

Skript v MATLABu vytvoříme tak, že si nejdříve nastavíme pracovní adresář. Nový M-soubor, který bude obsahovat skripty, vytvoříme například pomocí hlavního menu File → New → M-file, čímž se také otevře okno M-editoru. Do prázdného M-souboru zapíšeme všechny příkazy, které má skript provést.

Příkazy píšeme stejně jako v Command Window, jen s tím rozdílem, že se po napsání neprovádějí. Takto vytvoříme kód skriptu (posloupnost příkazů).

Máme-li napsaný kód skriptu, musíme celý M-soubor uložit pod nějakým názvem na disk do pracovního adresáře. Uložení provedeme pomocí menu File → Save, kde zkontrolujeme pracovní adresář a zadáme název skriptu. Jméno skriptu musí splňovat stejná pravidla jako název proměnné, přestože se jedná o jméno souboru. Jméno M-souboru se skriptem tedy může obsahovat pouze písmena anglické abecedy, podtržítko a číslice, ale číslicí nesmí začínat.

Spuštěním skriptu dáváme MATLABu pokyn k vykonání jeho příkazů. Před spuštěním musí být skript uložen. Skript můžeme spustit buď v Command Window a to tak, že stačí zadat název M-souboru bez přípony (.m), anebo v M-editoru/Debuggeru pomocí menu Debug→Run.



Obr. 8 Vykreslení grafu po spuštění skriptu

Je-li MATLAB ukončen, všechny proměnné definované při práci se ztrácí. Vyvoláním příkazu save před ukončením MATLABu dojde k zapsání všech proměnných do diskového souboru matlab.mat. Při dalším otevření MATLABu mohou znovu vyvolány příkazem load.

4.2 VBScript (Visual Basic – Scripting Edition)

Visual Basic Scripting Edition (dále VBScript či VBS) je skriptovací jazyk, který se vyvinul z programovacího jazyka Visual Basic.

VBScript je po JavaScriptu nejpoužívanější skriptovací jazyk hlavně díky podobnosti Visual Basicu a téměř neomezeným schopnostem. Skript nelze samostatně spustit. Musí být načten nějakým programem a převeden (interpretován) do podoby, kterou operační systém přijme. Tento převod zajišťuje u VB skriptů a Java skriptů tzv. Scripting Engine, což jsou pro účely WSH soubory wscript.exe a cscript.exe, které jsou standardním vybavením Windows.

Za pomocí VBScriptu můžeme docílit efektivní práce se systémem Windows (kopírování souborů, automatické zálohování, vymazávání, instalace), automatizovaně spravovat web (upload, příprava stránek k publikaci, čištění špinavého wordovského HTML), spravovat databáze, ulehčit si hodiny práce při správě sítě, skriptovat na straně serveru i na straně klienta (u skriptování na straně klienta jsou ale možnosti omezené, tam je lepší použít jazyk JavaScript) atd.

4.2.1 Vytváření skriptů

VBScript se může zapisovat do běžných textových souborů (např. v Poznámkovém bloku), které se uloží s příponou VBS. Takto uložený skript se pak spouští pouhým stiskem klávesy enter a Windows Script Host se už postará o překlad a provedení příkazů.

Dokonce lze skript i spouštět a mít ho přitom otevřený v editoru a dolaďovat kód před každým spuštěním. Při spuštění se použije poslední uložená verze.

VBScript dokáže pohodlně pracovat s objekty a jejich metodami a vlastnostmi. Při použití objektů se musíme na ně odkazovat. Existují dva druhy odkazů. Ty, které již jsou součástí WHS, a ty, které si vytváříme. Abychom jej mohli vytvořit, potřebujeme proměnnou, která si bude tento odkaz pamatovat. Když chceme získat odkaz na objekt, použijeme příkaz Set. Objekty umožňují přístup k jádru Windows, registrům a stavovým proměnným. Lze jej využít ke spouštění programů, vytváření zástupců, zobrazování zpráv v dialogových oknech, přístupu k systémovým složkám atd.

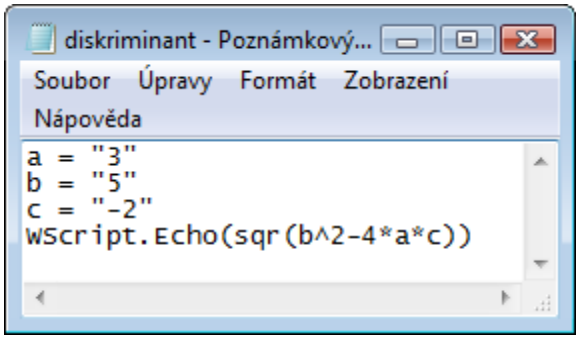
4.2.2 Windows Script Host (WSH)

Windows Skript Host využívá prostředky jazyka VBScript a JavaScript. Jejich používání ve WSH se tak stalo plnohodnotným skriptovacím prostředím pro řízení operačního systému Windows a k plnění užitečných úloh. Dokonale automatizují činnost uživatele při obsluze Windows a dalších aplikací a podstatně urychlují práci a orientaci uživatele. Jedním, dvěma kliknutími spustíte skript, který za vás provede úlohu, jež při ručním ovládání zabere mnohem více času. Přitom skripty jsou spolehlivé, chrání uživatele před chybami a sdělí mu, kde se dopustil nepřesnosti. Vedle běžných činností (zjišťování, otevírání, přemísťování a kopírování souborů, čtení a zapisování dat) umožňuje WSH:

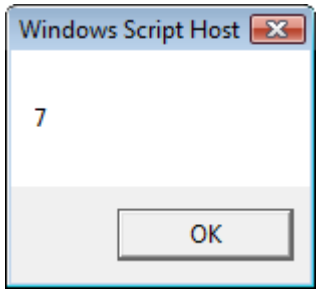
1. Rychlé zpřístupnění příkazů operačního systému.
2. Snadné nastavení operačního systému.
3. Možnost správy uživatelských jmen.
4. Spolupracuje s aplikacemi Word, Excel. Může automatizovat správu Microsoft SQL Serveru a IIS.
5. Umožňuje využití komponent ActiveX.
6. Spolupracuje s MS Outlook Expresem.

Skripty můžeme psát v jednoduchém textovém editoru (Poznámkový blok), ale nejlepší bude použít některý editor HTML, který čte a také zobrazuje soubory vbs, např. freewarový 1st Page 2000. Použít můžeme rovněž textové editory MS Word (má zabudovaný editor Visual Basic), nebo MS Office (obsahuje Script Editor), či v neposlední řadě český programovací editor PSPad.

WSH se dá také využít pro jednoduché výpočty a to tak, že si v Poznámkovém bloku napíšeme potřebný vzorec a uložíme ho s příponou .vbs pro jazyk VBSkript a nebo .js pro JSkript. Scripting Engine přeloží soubor a vlastnost Echo vypíše výsledek.



Obr. 9 Skript napsaný v Poznámkovém bloku



Obr. 10 Výsledek skriptu

4.3 Návrh svého skriptovacího jazyka

Z počátku jsem ve své praktické části začal vytvářet svůj skriptovací jazyk úplně od začátku podle zásad a pravidel, které jsou popsané v úvodních dvou kapitolách mé práce. Nicméně po několika konzultacích s odbornými asistenty naší katedry, jsem se přiklonil k řešení nevytvářet nový skriptovací jazyk úplně od začátku, ale použít hotový ActiveX prvek Microsoft Script Control, který poskytuje skriptovací prostředí v jazyce VBScript a ten rozšířit o vlastní nové funkce potřebné pro matematické a vědecké výpočty. Navržený skriptovací jazyk musí umět zápis příkazů pro výpočet vzorců nebo celých algoritmů obsahujících i podmíněné příkazy a cykly.

4.3.1 Výrazy a proměnné ve vzorcích

Výraz je posloupnost konstant, názvů proměnných, operátorů (včetně kulatých závorek) a volání funkcí. Výsledek výrazu můžeme uložit buďto do nějaké proměnné, pokud chceme dále s touto hodnotou pracovat, nebo můžeme hodnotu vypsat na obrazovku. K tomu nám slouží buď standardní příkaz MsgBox, který vypíše výsledek na obrazovku pomocí dialogového okna nebo pomocí přidaných funkcí WScalar pokud jde o skalár, WVector pokud jde o vektor a WMatrix pokud jde o matici. Pro výpočet vzorce bez proměnných a výpis rychlého výsledku na obrazovku můžeme také využít metody Eval(), která se vykoná po zmačknutí tlačítka Eval.

Název proměnné může obsahovat až 31 znaků a to písmena anglické abecedy (a-z, A-Z), číslice (0-9) a podtržítko (_). Číslicí název začínat nesmí. V názvech nejsou rozlišována velká a malá písmena. Skriptovací jazyky nerozlišují různé typy proměnných.

K sestavení vzorce lze použít následující aritmetická znaménka: +, -, *, /, ^ (mocnina), Sqr (odmocnina), zřetězení &, relační znaménka == (rovnost), <> (nerovnost), <, >, <=, >=, logické operátory Not, And, Or, Xor, Eqv. Závorky určují prioritu operátorů, pokud má být jiná, než je výchozí.

4.3.2 Podmíněné příkazy

Kód se vykonává v určitém pořadí, tomu se často říká tok. V jednoduchých skriptech se vykonává shora dolů, v těch složitějších bývá větvení, cyklování

a volání funkcí a procedur. Cykly jsou segmenty kódu, které slouží k tomu, aby se něco vykonávalo opakovaně, např. dokud platí nebo neplatí nějaká podmínka.

Do … Loop

Cyklus určený pro opakování bloku příkazů v závislosti na splnění podmínky. Má dvě formy. Jedna dává podmínku, za které se má kód vykonávat, k Do (tam, kde např. chceme, aby za určitých okolností cyklus vůbec neproběhl), druhá k Loop. Podmínka se konstruuje pomocí slova While či Until. While říká: vykonávej, dokud platí podmínka. Until říká: vykonávej, dokud nezačne platit podmínka.

Tab. 3 Syntaxe podmíněného cyklu Do … Loop

Do [{While | Until} podmínka]	***Do***
[příkazy]	*[příkazy]*
[Exit Do]	***[Exit Do]***
[příkazy]	*[příkazy]*
Loop	***Loop [{While | Until} podmínka]***

If ... Then ... ElseIf ... Else ... End If

If podmínka Then ... [ElseIf] ... [Else] ... End If je nejběžnější větvení určené k vykonání bloku kódu za splnění určité podmínky. ElseIf a Else jsou volitelné. End If se naopak v blokovém If vynechat nesmí. Za ElseIf se může specifikovat další podmínka. Za Else se dává, co se má udělat, pokud není podmínka za If ani žádným z ElseIf splněna.

Tab. 4 Syntaxe If … Then … ElseIf … Else … End If

If podmínka Then
[příkazy]
ElseIf podmínka 2
[příkazy]
Else
[příkazy]
End If

Jednořádkové If ... Then se neukončuje pomocí End If – to je jen pro blokové If. V jednořádkovém If ... Then lze použít i Else. Naopak není možné v jednořádkovém zápisu použít ElseIf.

4.3.3 Cykly

For ... Next, For Each ... Next

Cyklus For ... Next je určen k vykonání bloku kódu pro každý člen určité množiny. Tuto množinu buď můžeme mít předem (může jí být například kolekce všech souborů v adresáři), nebo ji můžeme vytvořit přímo na začátku cyklu. Pokud pracujeme s kolekcí, použijeme For Each ... Next. Pokud chceme stanovit číselný rozsah, použijeme For ... To ... Next. Vyskočení z cyklu se provede příkazem Exit For.

Tab. 5 Syntaxe cyklu For … To … Next

For i = cislo1 To cislo2
[příkazy]
Next

Příklad cyklu For: Ten to příklad spočítá, kolik prvků pole je čistě číselných.

```
Pole = array(-5, 12, "a", 0)
pocetCiselnych = 0
For i = 0 to UBound(pole)
    If IsNumeric(pole(i)) Then pocetCiselnych = pocetCiselnych + 1
Next
MsgBox pocetCiselnych
```

Výsledek: 3

4.4 Komponenta MSC rozšířená o vlastní funkce

Použití komponenty MSC je popsáno v páté kapitole. Teď si ukážeme, o jaké funkce jsem MSC rozšířil.

Tato rozšířená komponenta má převážně pracovat se vzorky signálů, matematicky řečeno s vektory případně nebo maticemi. Visual Basic a tudíž VBScript „neumí" pracovat s vektory a maticemi, takže jsem rozšířil veškeré matematické funkce VBScriptu o práci s vektory a maticemi.

Při tvorbě nových funkcí jsem dbal na jednoduchost používání skriptovacího jazyka. Názvy funkcí jsem zvolil zkráceně pro rychlejší psaní skriptu a v anglickém jazyce, stejně jako je psaný VBScript. Veškeré funkce provádějí operace s patřičným počtem argumentů (proměnných). Funkce, které vracejí nějakou hodnotu výsledku, musí být přiřazeny proměnné. Seznamy funkcí a jejich syntaxe jsou znázorněné v tabulkách následujících podkapitol.

4.4.1 Aritmetické operace s vektory a maticemi

Vytvořil jsem funkce, které provádí aritmetické operace s vektory a maticemi a to tak, že např. při sčítání dvou vektorů se sečtou navzájem všechny prvky stejných indexů. Pokud zadáme nestejné délky signálů, vypíše se na obrazovku chybové hlášení. Funkce je navržena tak, že může provádět operaci se dvěma argumenty.

Tab. 6 Názvy aritmetických funkcí

Aritmetické operace	**S vektory**	**S maticemi**
Sčítání	*SumVec(arg1, arg2)*	*SumMat(arg1, arg2)*
Odečítání	*DifVec(arg1, arg2)*	*DifMat(arg1, arg2)*
Násobení	*ProdVec(arg1, arg2)*	*ProdMat(arg1, arg2)*
Dělení	*PartVec(arg1, arg2)*	

4.4.2 Matematické funkce

Matematické funkce patří mezi standardní funkce jazyka VBScript, takže jsem je rozšířil tak, aby prováděli patřičné operace s vektory i maticemi. Funkce provádí operace s jedním argumentem (proměnou), kterému je přiřazen skalár, vektor nebo matice.

Tab. 7 Názvy matematických funkcí

Matematické operace	**S vektory**	**S maticemi**
Odmocnina	*SqrVec(arg1)*	*SqrMat(arg1)*
Exponenciála	*ExpVec(arg1)*	*ExpMat(arg1)*
Logaritmus	*LogVec(arg1)*	*LogMat(arg1)*
Absolutní hodnota	*AbsVec(arg1)*	*AbsMat(arg1)*

Signum	*SgnVec(arg1)*	*SgnMat(arg1)*
Zaokrouhlení	*RoundVec(arg1)*	*RoundMat(arg1)*
Náhodné číslo	*RndVec(arg1)*	*RndMat(arg1)*

4.4.3 Goniometrické funkce

Goniometrické funkce taktéž patří mezi standardní funkce jazyka VBScript. Znovu jsem rozšířil tyto funkce, aby byli schopny provést goniometrické operace s vektory a maticemi. Tyto funkce mohou provádět operace pouze s jedním argumentem (proměnou), kterému je přiřazen vektor nebo matice.

Tab. 8 Názvy Goniometrických funkcí

Goniometrické operace	**S vektory**	**S maticemi**
Sinus	*SinVec(arg1)*	*SinMat(arg1)*
Cosinus	*CosVec(arg1)*	*CosMat(arg1)*
Tangens	*TanVec(arg1)*	*TanMat(arg1)*
Arkus tangens	*AtnVec(arg1)*	*AtnMat(arg1)*

4.4.4 Funkce pro načtení vektorů a matic a pro vypsání výsledků

Funkce pro načtení vektorů čili pole se jmenuje Array a je to standardní funkce VBScriptu. V tomto případě jsem nemusel rozšiřovat tuto funkci, ale nahradil jsem tento název za dvě hranaté závorky [ ]. Např. Místo *A = array(1,2,3)* zapíšu pouze *A = [1,2,3].* Ale je to nepovinné, klidně se může používat i název array().

Při načítání matice už je to značně složitější. Tam jsem musel provést vlastní funkci, která načítá dvě pole, a to pole hodnot v řádcích a pole hodnot ve sloupcích. Tuto funkci GetMat jsem opět nahradil hranatými závorkami. Jednotlivé prvky oddělujeme čárkou po sloupcích a středníkem po řádcích.

Pro výpis výsledků jsem vytvořil také vlastní funkce a to hned čtyři. Záleží totiž na tom, jaký typ výsledku chceme vypsat. Může to být skalár, vektor, matice anebo pouhý text (string) a pro každé je jiná funkce. Dále jsem vytvořil funkci,

která načte data do MS Excelu a vykreslí graf hodnot. Seznam těchto funkcí a syntaxe je znázorněna v následující tabulce.

Tab. 9 Seznám funkcí pro načtení a výpisy výsledků skalárů, vektorů a matic

Seznam funkcí	**„Preprocesorink“**	**Názvy funkcí**
Načtení vektoru	*[ ]*	*Array()*
Načtení Matice	*[ ]*	*GetMat()*
Výpis skaláru		*WScalar(arg1)*
Výpis vektoru		*WVector(arg1)*
Výpis matice		*WMatrix(arg1)*
Výpis textu		*WText(arg1)*
Vykreslení grafu v Excelu		*GraphExcel(arg1)*

4.4.5 Filtry pro zpracování signálu

Filtry v oboru zpracování signálů jsou zařízení k úpravě frekvenčního spektra signálu. Princip jejich funkce může být založen na analogovém obvodu nebo algoritmu číslicového počítače.

Číslicové filtry nepracuji v čase spojitě, ale jejich výstup je dán periodickým výpočtem ze změřených vzorků signálu. Filtrovat lze ve frekvenční nebo v časové oblasti. Ve frekvenční oblasti se filtruje násobením frekvenčního přenosu filtru obrazem vstupního signálu, který je třeba v průběhu měření opakovaně vypočítat, přičemž následuje inverzní transformace zpět do časové oblasti. Tento způsob filtrace je vhodný jen pro dodatečnou úpravu signálů.

Tab. 10 Seznám funkcí filtrů

Seznam filtrů	**Názvy funkcí**
LowPassFilter	*LowPassFilter(s, n, f)*
HighPassFilter	*HighPassFilter(s, n, f)*
DifferentiatorFilter	*DifferentiatorFilter(n, f)*
HilbertTransformer	*HilbertTransformer(n, f)*
BandPassFilter	*BandPassFilter(n, f_a, f_b)*

Tyto funkce vracejí pouze koeficienty filtrů. Následující tabulka funkcí zpracovává signál respektive vektor hodnot s předešlými funkcemi filtrů. Do těchto funkcí se za argumenty zadá signál hodnot, typ filtru nebo koeficienty filtrů a popřípadě ještě některé další argumenty a tato funkce vyfiltruje signál, který jsme zadali jako první argument.

Tab. 11 Seznám funkcí pro filtrování signálů

Seznam funkcí	**Názvy funkcí**
SignalFilter	*SignalFilter(DataVector, FilterType, Odder, f1, f2)*
SignalFilter2	*SignalFilter2(FiltrCoef,DataVector)*

4.4.6 Fourierova transformace

Funkce byla vytvořena k výpočtu rychlé Fourierovy transformace (FFT) pro počet reálných vzorků rovný mocnině dvou. Argument InputData obsahuje reálnou složku vstupních dat, přičemž imaginární složka se předpokládá nulová. Výsledkem výpočtu je signál o dvojnásobné délce ve srovnání se vstupním signálem. První polovina počtu vzorků představuje reálnou složku výsledku výpočtu a druhá polovina představuje imaginární složku.

Tab. 12 Seznám funkcí pro výpočet FFT

Seznam funkce	**Název funkce**
FFT	*FFTW(InputData)*

4.5 Příklady použití předchozích funkcí

Př: 1. Logaritmus součtu dvou vektorů:

```
a = [1,2,3,4,5]
b = [10,11,12,13,14]
c = SumVec(a,b)
d = LogVec(c)
WVector(d)

Výsledek:  2,3979  2,5649  2,7081  2,8332  2,9444
```

Př: 2. Sinus rozdílu vektoru s konstantou a signum výsledného sinusu:

```
a = [3]
```

```
b = [10,11,12,13,14]
c = DifVec(b,a)
d = SinVec(c)
WVector(d)
e = SgnVec(d)
WVector(e)
```

Výsledek: 0,657 0,9894 0,4121 -0,544 -1
1 1 1 -1 -1

Př: 3. Z rozdílu dvou matic spočtena absolutní hodnota a z té potom odmocnina:

```
a = [0,0.5,1;1.5,2,2.5;3,3.5,4]
b = [10,11,12;13,14,15;16,17,18]
c = DifMat(a,b)
d = AbsMat(c)
e = SqrMat(d)
WMatrix(e)
```

Výsledek: 3,1623 3,2404 3,3166
3,3912 3,4641 3,5355
3,6056 3,6742 3,7417

Př: 4. Do proměnné b jsme uložili koeficienty filtru o 9 prvcích a frekvencí 0,8 a pak jsme použili funkci SignalFilter2 pro filtraci signálu:

```
a = [10,11,12,13,14,15]
b = IdealDifferentiatorFilter(9,0.8)
d = SignalFilter2(a,b)
WVector(d)
```

Výsledek: 8,2535 0,6468 0,1399 1,3591 3,0954 -10,1556

5 Microsoft Script Control

Microsoft Script Control (MSC) je ovládací prvek ActiveX, který poskytuje aplikacím skriptovací prostředí. Smyslem skriptovacího jazyka je poskytnout uživateli možnost doprogramovat vlastní algoritmy a rozšířit aplikaci o další potřebné funkce.

5.1 Použití modulů a kolekcí procedur

Jak bylo řečeno, MSC poskytuje aplikacím skriptovací prostředí. Podporované skriptovací jazyky pro tento ovládací prvek jsou VBScript, JavaScript a některé další kompatibilní jazyky, které pracují s internetovými prohlížeči. MSC dovoluje aplikacím vkládat části skriptů do různých modulů. Různé moduly mohou obsahovat procedury stejných názvů.

Tato komponenta také obsahuje vlastnosti procedur, které dovolují aplikacím dynamicky určit, zda procedura je funkce nebo subrutina stejně jako očekávané číslo argumentu, které vybídne uživatele k zadání správného počtu hodnot parametrů.

Vlastnosti modulu Script control jsou objekty, který obsahují kolekci objektů modulu. To má následující vlastnosti a metody:

- Count: Číslo modulu
- Item(x): Vrací jediný objekt modulu
- Add name: Přidává prázdný modul s daným názvem

Objekt modulu má následující vlastnosti a metody:

- Name: Název modulu
- Procedures: Procedury objektu
- AddCode code: Přidává kód do modulu
- Run name, args: Běžící pojmenovaná procedura

Objekt procedury obsahuje kolekci objektů procedury. To má následující vlastnosti a metody:

- Count: Číslo procedur v modulu
- Item(x): Vrací jediný objekt Procedury

Objekt procedury má následující vlastnosti a metody:

- Name: Název procedury
- HasReturnValue: Ukazuje, zda procedura je subrutina (Sub) nebo funkce (Function)
- NumArgs: Procedura vyžaduje číslo argumentu

5.2 Metody a volání funkcí

Tato podkapitola popisuje metody, které nám nabízí ovládací prvek Script Control a volání používaných funkcí.

Script Control poskytuje tyto čtyři metody pro volání subrutin a funkcí (Sub a Function):

- Eval: Vyhodnotí text výrazu
- Run: Spouští pojmenovanou Sub nebo Function
- Execute: Vykonává příkazový skript
- Metoda objektu modulu

5.2.1 Eval

Volající konvence:

```
Vysledek = ScriptControl.Eval(„nejaký textový výraz“)
```

Tato metoda se používá k volání vnitřní skriptovací funkce, stejně jako uživatelské funkce. Argumenty funkce jsou zadány jako text ve výrazu a můžou být buď zapsané v kódu programu, anebo zřetězeny s proměnnou. Metoda nemůže být použita k volání subrutin ani funkcí.

5.2.2 Run

Volající konvence:

```
ScriptControl.Run(“Name“, arg1, arg2, … argn)
```

Tato metoda slouží k volání subrutin, kde je napsaný celý skript. Při načítání skriptu se celá subrutina načte pomocí vlastnosti AddCode a poté je spuštěna metoda Run, která zavolá překlad skriptu.

Name je název subrutiny nebo funkce a arg1 … argn jsou nepovinné argumenty závisející na subrutině nebo funkci.

5.2.3 Execute

Volající konvence:

```
ScriptControl.Execute "nějaký text"
```

Metoda Execute umožňuje volat nějaké skutečné příkazy nebo subrutiny. Můžete ji také použít k zavolání funkcí, ale vrácený výsledek je zahozen.

5.2.4 Module

Volající konvence:

```
Vysledek = ScriptControl.Modules(modulename).functionname(arg1, arg2, …)
ScriptControl.Modules(modulename).subname arg1, arg2, …
```

Standardní modul GlobalModule je daný konstantou, například:

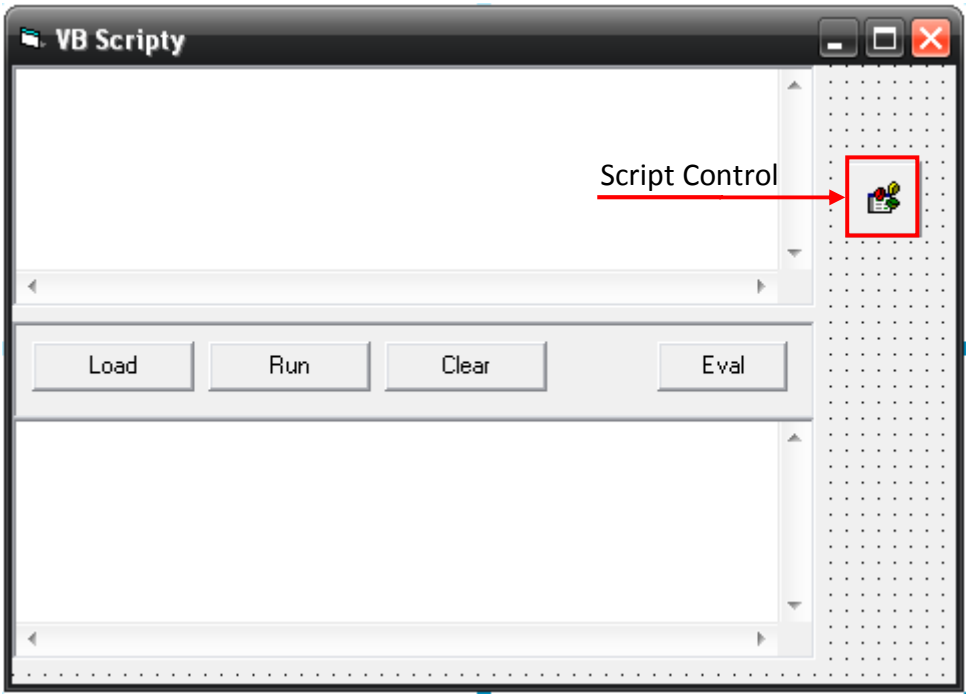
```
Vysledek = ScriptControl.Modules(GlobalModule).Myfunction(5)
ScriptControl.Modules(GlobalModule).MySub 5, "A"
```

6 Návrh software pro výpočet skriptu

Software pro výpočet skriptu řešící zápis příkazů pro výpočet vzorců nebo celých algoritmů obsahujících podmíněné příkazy a cykly, spočívá v komponentě Microsoft Script Control (MSC). Smyslem skriptovacího jazyka je poskytnout uživateli možnost doprogramovat vlastní algoritmy. Pomocí jazyka VBScript máme téměř neomezené možnosti v psaní skriptu a navíc máme k dispozici vlastní funkce, o které jsem MSC rozšířil.

6.1 Návrh aplikace pro výpočet skriptu

Díky komponentě MSC není třeba programovat složité aplikace, ale stačí ve Visual Basicu založit nový projekt, na formulář přesunout dvě textová pole, několik tlačítek, kterým nadefinujeme jednoduché metody a ActiveX ovládací prvek Script Control. Pokud tento prvek nevidíme na panelu s ovládacími prvky, klikneme na Projekt v hlavní nabídce menu a vybereme položku Components. Tam zaškrtneme políčko “Microsoft Script Control 1.0“ a potvrdíme tlačítkem OK. Poté se objeví v Toolboxu ovládací prvek ScriptControl, který umístíme kdekoliv na pozadí formuláře. Návrh formuláře může vypadat například jako na obr. 13.



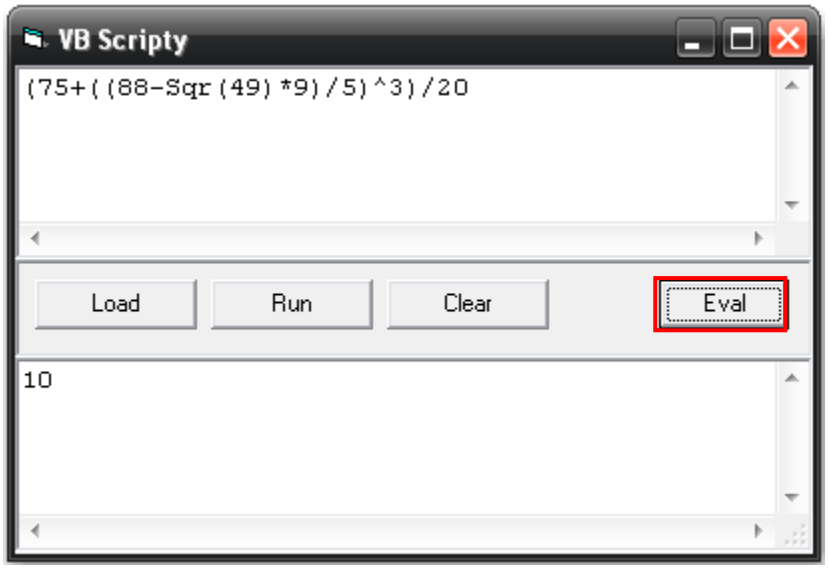
Obr. 13 Příklad návrhu formuláře pro výpočet skriptů s využitím Script Control

6.2 Použití metod a vlastností MSC pro výpočet skriptu

Abychom byli schopni používat MSC, který poskytuje skriptovací prostředí, musíme kromě přidání Script Control do formuláře také nadefinovat tlačítkům následující metody.

Pro tlačítko Eval použijeme metodu Eval(), která slouží pro rychlý výpočet vzorců, které zapisuje do výsledného textového pole. Syntaxe metody je následující.

```
Private Sub cmdEval_Click()
   txtOutput.Text = ScriptControl.Eval(txtSource.Text)
End Sub
```



Obr. 14 Příklad výpočtu vzorce metodou Eval

Tlačítko Load nám pomocí metody AddCode vkládá skripty do komponenty ScriptControl. Tento skript musí být součástí subrutiny, takže je dobré si zřetězit název subrutiny např. Sub Main() se skriptem a nakonec zase přidat End Sub. Následuje syntaxe.

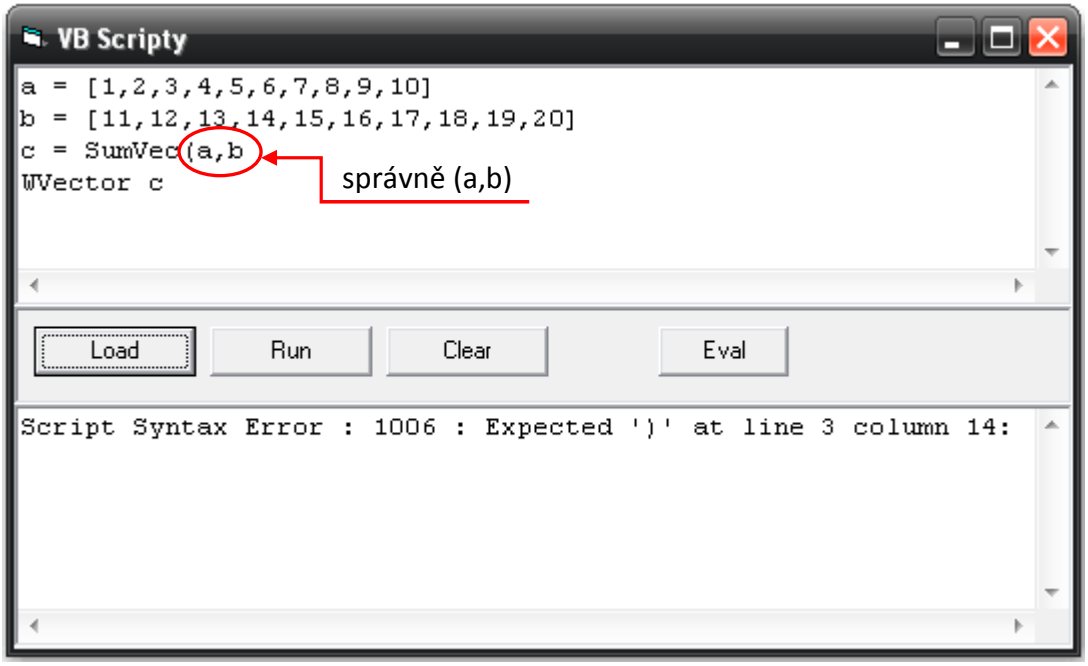
```
Private Sub cmdLoad_Click()
   Dim code As String
   code = "Sub Main()" & vbCrLf & txtSource.Text & vbCrLf & "End Sub"
   ScriptControl.AddCode code
End Sub
```

Také se tu dá používat nahrazování názvů funkcí, o které jsem rozšířil komponentu MSC, které vedou k jednoduššímu psaní skriptu.

```
code = Replace(code, "[", "GetMat(""")      'nahrazeni funkce GetMat()
code = Replace(code, "]", """)")            'za hranate zavorky
```

Je dobré zde využívat vlastnost Error, která slouží pro výpis chybových hlášení a nápovědy při kontrole syntaxe skriptu. Chybové hlášení nám vypíše, co vzniklo za chybu a kde přesně se vyskytuje (řádek a sloupec). V opačném případě vypíše bezchybné hlášení, že kontrola syntaxe skriptu proběhla v pořádku (No Load Errors).

```
If Err Then
  With scrMain.Error
    txtOutput.Text = "Script Syntax Error : " & .Number & " : " & .Description &
    " at line " & .Line & " column " & .Column & " : " & vbCrLf & txtOutput.Text
  End With
Else
  txtOutput.Text = "No Load Errors." & vbCrLf & txtOutput.Text
End If
```



Obr. 15 Výpis chybového hlášení při pokusu o načtení skriptu

Následující tlačítko Run spouští metodu Run, která provádí interpretaci skriptu, který byl načten metodou AddCode po zmáčknutí tlačítka Load. Tato metoda provádí rozklad libovolných výrazů a posloupnosti příkazů pro výpočet vzorců a celých algoritmů obsahujících podmíněné příkazy a cykly. Dále umí pracovat s poli a objekty. Za metodu Run se pouze napíše do uvozovek název subrutiny, v našem případě "MAIN". Syntaxe je následující.

```
Private Sub cmdRun_Click()
  ScriptControl.Run "MAIN"
End Sub
```

Též je vhodné u této metody používat vlastnost Error, která vypisuje chybové hlášení při interpretaci skriptu. V případě, že překlad proběhl v pořádku, je vypsán výsledek. Poslední tlačítko Clear, nám pouze vyčistí výsledné pole.

6.3 Vytváření vlastních funkcí používaných v MSC

Krása používání komponenty MSC nespočívá pouze v její jednoduchosti a zároveň síle, ale i v její kompatibilitě širokého použití a hlavně v možnosti rozšíření o vlastní funkce.

V předchozí kapitole jsme si ukázali, jak se používají metody a vlastnosti v MSC. Teď si ukážeme, jak rozšířit tuto komponentu o vlastní funkce. Abychom mohli používat ve skriptech vlastní naprogramované funkce, musíme nové funkce vytvářet ve třídách (Class Modulu) a deklarovat je jako veřejné (Public).

Následuje ukázka funkce *GraphExcel(arg1)* o kterou jsem rozšířil komponentu MSC. Vstupním argumentem této funkce je proměnná, ve které je uložen vektor s daty. Po spuštění skriptu se spustí aplikace MS Excel, do listu se uloží data a vykreslí se graf. Zdrojový kód funkce pro tuto operaci vypadá následovně.

```
Public Function Graph(A())
   Dim EX, Book, Sheet, Source, PlotBy
   Dim i As Long
   Dim Str As String
   Set EX = CreateObject("Excel.Application")
   EX.Visible = True
   Set Book = EX.Workbooks.Add
   Set Sheet = Book.Sheets(1)
   For i = 0 To UBound(A)
      EX.ActiveSheet.cells(i + 1, 1) = A(i)
   Next
   Str = "A1:A" & CStr(UBound(A) + 1)
   EX.ActiveSheet.Range(Str).Select
   EX.Charts.Add
   EX.ActiveChart.ChartType = 4
End Function
```

Samozřejmě, že jde použít tuto operaci i bez vytváření funkce, ale museli bychom podobný zdrojový kód psát pokaždé do skriptu, což je značně nepříjemné, zdlouhavé a složité. Místo toho stačí do skriptu zadat název funkce s názvem proměnné GraphExcel(NázevProměnné).

Tyto třídy kde jsou vytvořené vlastní funkce, musíme deklarovat ve formuláři jako objekty.

```
Dim fftw As New FFT          ’ třída, která provádí vypočty FFT
Dim fil As New FilterDesign  ’ třída, která provádí výpočty s filtry
Dim vec As New Vector        ’ třída, která provádí výpočty se signály
Dim mat As New Matrix        ’ třída, která provádí výpočty s maticemi
```

Potom se tyto objekty načítají při každém spuštění formuláře do komponenty MSC pomocí metody AddObject.

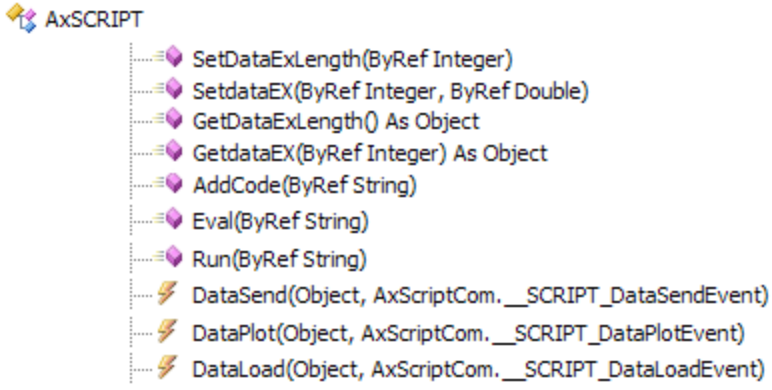
```
Private Sub Form_Load()
  ScriptControl.AddObject "FFT", fftw, True
  ScriptControl.AddObject "Filter", fil, True
  ScriptControl.AddObject "Vector", vec, True
  ScriptControl.AddObject "Matrix", mat, True
End Sub
```

Teď už můžeme používat nově přidané funkce. Stačí jen zapsat názvy funkcí s patřičnými parametry do zdrojového skriptu.

6.4 Vytvoření vlastní komponenty pro využití skriptů v reálných aplikacích

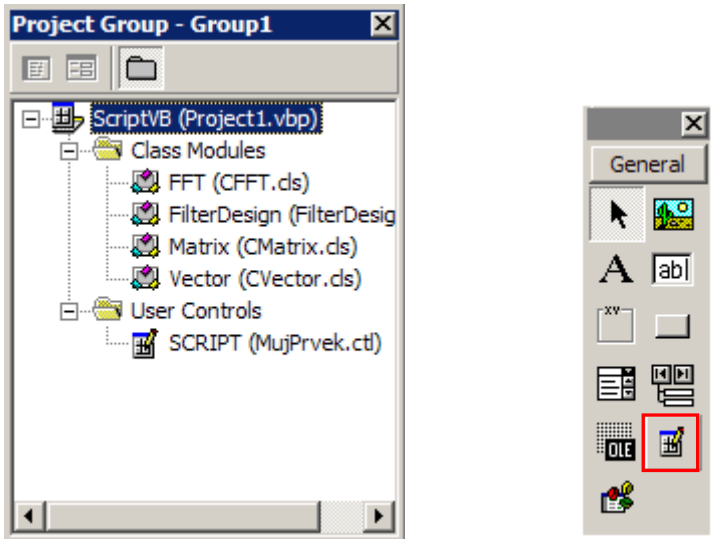
Aby bylo možné využívat Microsoft Script Control (MSC) rozšířený o vlastní funkce v reálných aplikacích, je zapotřebí přidat tyto funkce do komponenty MSC, zkompilovat ji a vytvořit tak vlastní ActiveX komponentu. Vzniklá komponenta má tak v sobě implementované nové funkce.

Dále je ještě před kompilací komponenty zapotřebí, domluvit se s vlastníkem reálné aplikace na společném rozhraní, které nám zajistí správné a bezchybné propojení a přenos dat.



Obr. 16 Metody a události propojující komponentu s aplikací

Po vzájemné domluvě s autorem reálné aplikace bylo vytvořeno několik nových metod, událostí a pomocných polí, sloužících ke komunikaci a předávání dat mezi reálnou aplikací a novou komponentou (rozšířenou o vlastní funkce).



Obr. 17 Nová ActiveX komponenta rozšířená o vlastní funkce

Zkompilovaná nová komponenta, se pak používá v aplikacích úplně stejně jako každý jiný ActiveX prvek.

6.5 Rozhraní komponenty a reálné aplikace

Předešlé metody a události z obr. 16 jsou užity vždy, když chceme pracovat s komponentou v aplikaci. Následující metody a události se týkají pouze použití určitých funkcí ve skriptu.

Načtení naměřeného signálu z aplikace do proměnné Vstup:

```
Vstup = inputdata("NazevSignalu")
```

Při zpracování uvedeného výrazu je v komponentě zavolána funkce inputdata. Ta v první řadě vyvolá událost DataLoad, jejímž parametrem je název signálu, který chceme nahrát do komponenty. Tato událost je zachycena v aplikaci. Zde proběhne kontrola, zda daný signál v aplikaci existuje. Pokud ano, je připraveno pomocné pole, do kterého je uložen signál.

Následuje návrat do funkce inputdata, kde je signál z pomocného pole pomocí návratové hodnoty funkce předán proměnné Vstup. Takto nově vzniklá proměnná je typu array a je možné ji použít pod stejným názvem i v jiných částech

skriptu.

Funkce pro uložení proměnné z komponenty do aplikace:

```
OutputData1 Vystup, "NazevNovehoSignalu"
OutputData2 Vystup, SampleRate, "NazevNovehoSignalu"
```

Parametr Vystup, představuje proměnnou typu array, kterou chceme uložit do vlastní aplikace a "NazevNovehoSignalu" je jméno, pod kterým bude v aplikaci vystupovat. SampleRate udává jeho vzorkovací frekvenci. V případě OutputData1 je nastavena na frekvenci 1024. Při použití předcházejících dvou funkcí je komponentou volána metoda DataSend.

Funkce pro vykreslení signálu z komponenty:

```
Plot1 Value
Plot2 Value, SampleRate
```

Parametr Value představuje proměnnou typu array, kterou máme v úmyslu vykreslit a SampleRate udává stejně jako v předchozím případě vzorkovací frekvenci. Ta je u Plot1 nastavena na 1. Při použití předcházejících dvou funkcí je komponentou volána metoda DataPlot.

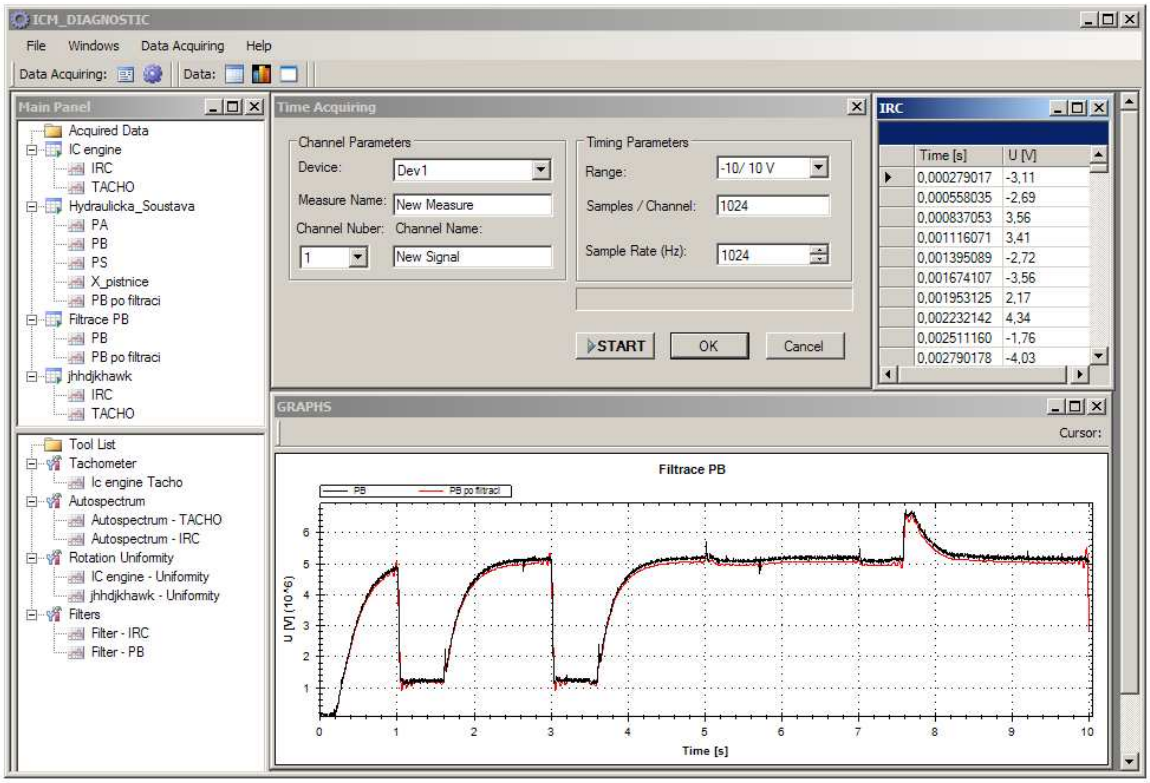
7 Použití rozšířené komponenty v reálné aplikaci

V této kapitole názorně ukážu využití rozšířené komponenty v reálné aplikaci a předvedu pár příkladů.

7.1 Reálná aplikace

Reálnou aplikací je diplomová práce zaměřena na problematiku diagnostiky spalovacích motorů na základě měření rovnoměrnosti otáčení klikového hřídele při ustálených volnoběžných otáčkách. Měření úhlové rychlosti probíhá pomocí IRC snímačů a Dopplerovského laseru. Autorem této práce je kolega Bc. Jan Benč.

Práce se skládá ze dvou hlavních částí. První část se zabývá návrhem měřicího řetězce a jeho hardwarovým vybavením. Druhá část se zabývá softwarovým zajištěním měřicí úlohy. Tvorbou vlastní měřicí aplikace pro sběr a zpracování naměřených dat.



Obr. 18 Reálná aplikace pro měření signálů a zpracování naměřených dat

Tato aplikace naměří vzorky signálu, což jsou data v podobě vektoru, které může dále pomocí vlastních nástrojů zpracovávat. Díky přidání mé komponenty se tak aplikace rozšíří o nespočetně dalších funkcí pro zpracování signálu.

7.2 Využití funkcí komponenty v reálné aplikaci

Tato komponenta je doplněna o funkce pro zpracování vektorů a matic tak, aby usnadnila práci uživatelům, kteří nemají zkušenosti v oblasti programovacího jazyka Visual Basic nebo VBScript (Visual Basic – Scripting Edition). Pro pokročilejší uživatele pak tato komponenta umožňuje s výhodou používat veškeré prvky, které jazyk Visual Basic nabízí. Tím jsou myšleny například podmiňovací příkazy jako If…Then, Do…Loop, while…Wend nebo Select case, dále cykly For…To…Next a For…Each…Next. Můžeme také využívat různé druhy operátorů a další prvky jako např. MsgBox, nebo InputBox. Dále je také možné vytvářet či přistupovat k objektům jiných aplikací, jako je Poznámkový blok, Word, Excel a mnoho dalších.

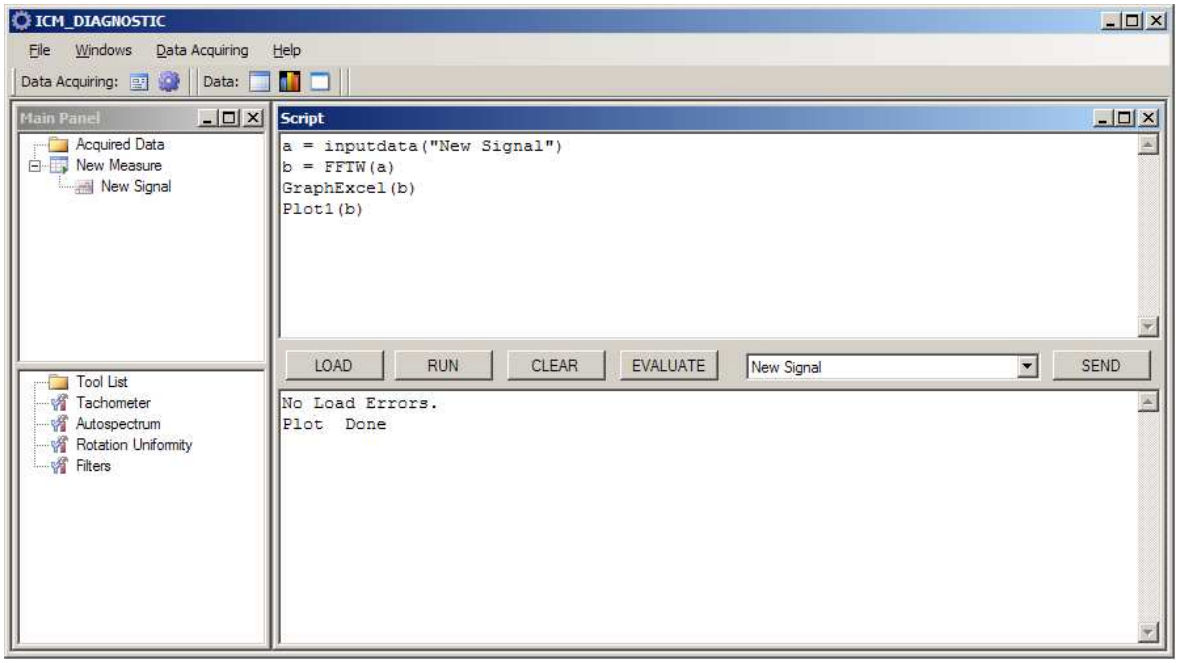
Díky široké paletě doprogramovaných funkcí může uživatel při zpracování vektorů a matic použít následující operace:

- Sčítání, odečítání, násobení a dělení
- Použití matematických funkcí: odmocnina, exponenciála, logaritmus, absolutní hodnota, signum, zaokrouhlení, náhodné číslo
- Použití goniometrických funkcí: sinus, cosinus tangens, arkus tangens
- Výpočet Furierovy transformace
- Návrh digitálních filtrů a jejich aplikaci na vstupní data
- Vykreslení dat v reálné aplikaci a v aplikaci MS Excel
- Možnost neustále doplňovat nové funkce

7.3 Příklady využití komponenty v reálné aplikaci

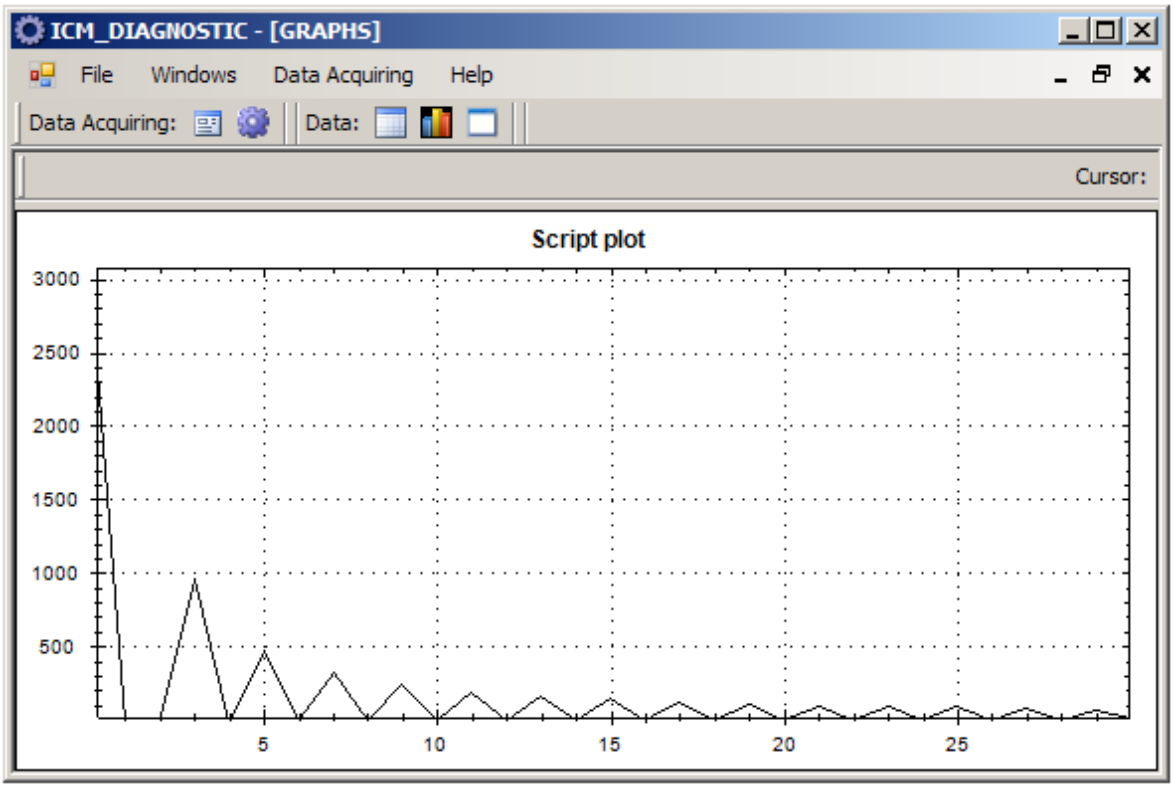
Nyní si předvedeme pár příkladů využití komponenty v reálné aplikaci:

Př. 1: Napřed naměříme data z měřicí karty. Tyto data vložíme do komponenty pomocí společné metody InputData(“New Signal“), kterou přiřadíme nějaké proměnné. Jak si proměnnou nazveme, není podstatné, v našem případě a. Potom provedeme výpočet diskrétní Fourierovi transformace s naměřených hodnot a necháme vykreslit grafy pomocí funkce GraphExcel a Plot1(b).



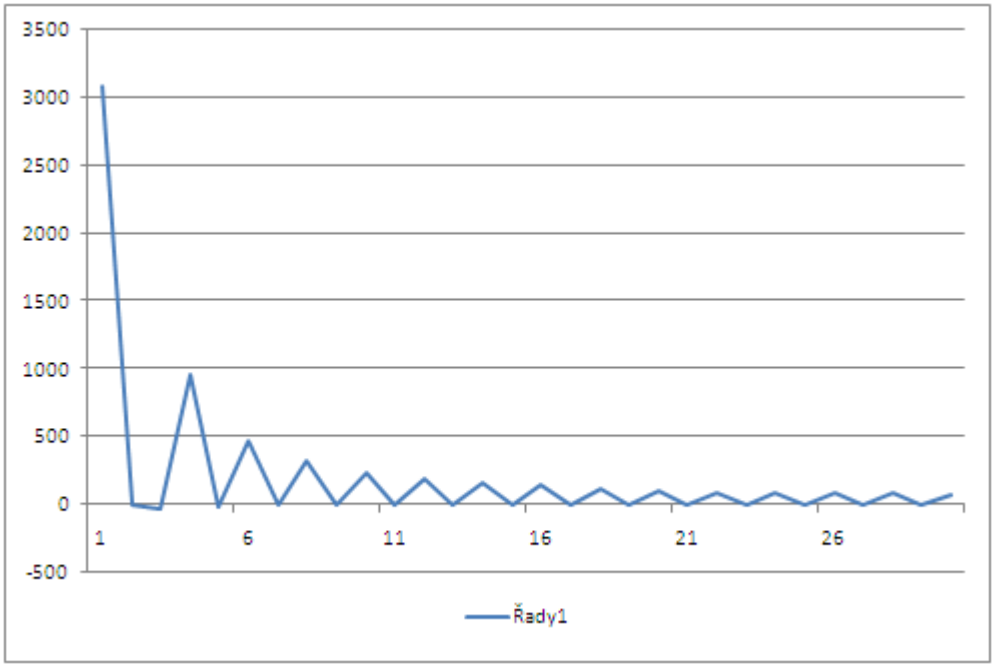
Obr. 19 Příklad využití komponenty v reálné aplikaci

Vykreslení grafu pomocí funkce Plot1 je v aplikaci zajištěno přidanou komponentou pod názvem ZedGraph. Data je možné zobrazit také v datové tabulce. Pomocí schránky je pak možné tyto data exportovat i do jiných aplikací.



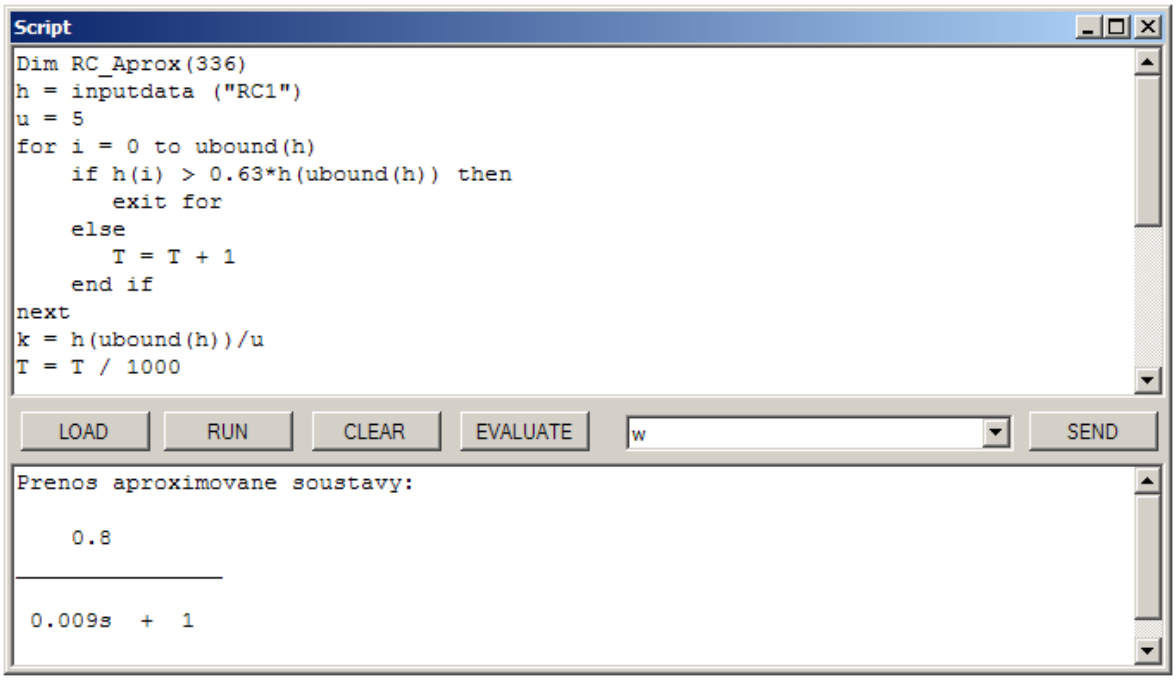
Obr. 20 Vykreslení grafu pomocí funkce Plot

Vykreslení grafu pomocí funkce GraphExcel v aplikaci MS Excel zajišťuje rozšířená komponenta, která pracuje i s objekty jiných aplikací.



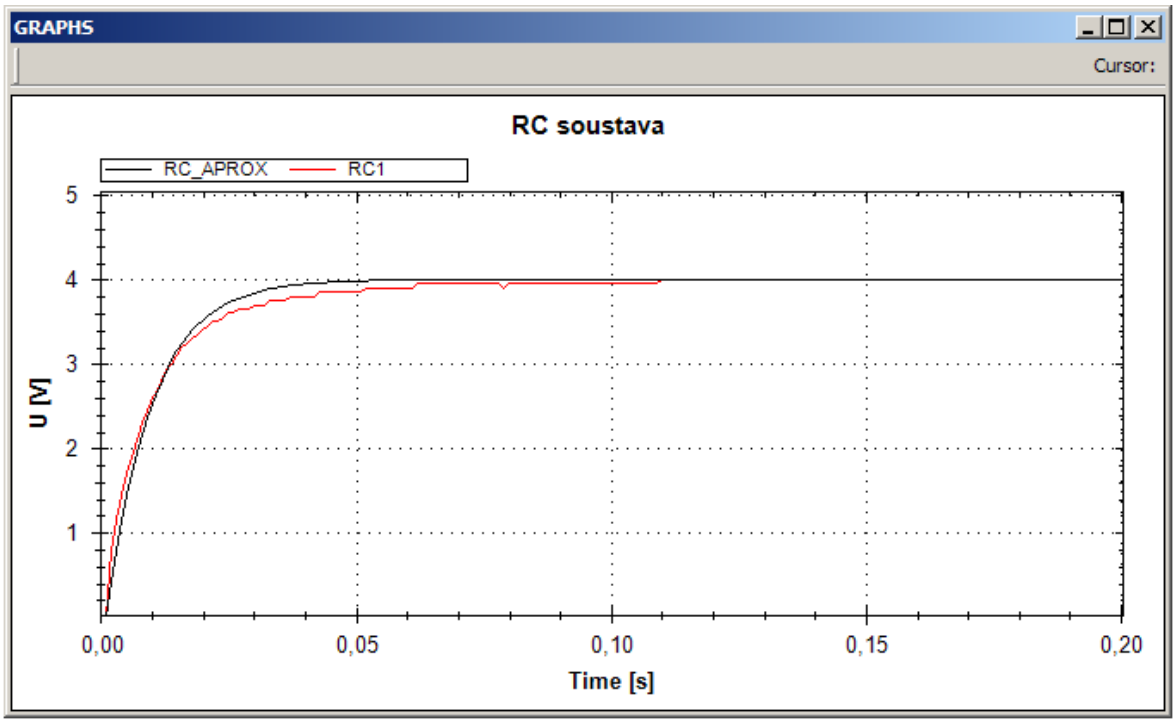
Obr. 21 Vykreslení grafu pomocí funkce GraphExcel v MS Excel

Př. 2: Následující příklad demonstruje využití skriptovacího jazyka v reálné aplikaci. Naprogramovaný skript provede aproximaci přechodové charakteristiky.



Obr. 22 Příklad skriptu aproximace přechodové charakteristiky v reálné aplikaci

Do výstupního okna se vypíše přenos aproximované soustavy a zobrazí se graf aproximované přechodové charakteristiky a původní přechodové charakteristiky.



Obr. 23 Aproximovaná a původní přechodová charakteristika

Skript aproximace přechodové charakteristiky z příkladu 2:

```
Dim RC_Aprox(336)
h = inputdata ("RC1")
u = 5
for i = 0 to ubound(h)
   if h(i) > 0.63*h(ubound(h)) then
      exit for
   else
      T = T + 1
   end if
next
k = h(ubound(h))/u
T = T / 1000
Wtext("Prenos aproximovane soustavy:")
Wtext("    " & cstr(k) & "    ")
Wtext("_________________")
Wtext(" " & cstr(T) & "s  +  1")

for i =0 to 336
RC_Aprox(i) = (1 - exp(-i/(T*1000)))*k*u
next
plot2 RC_Aprox, 1000
```

8 Zhodnocení výsledků a návrh směrů dalšího řešení

V této kapitole zhodnotím dosažené výsledky a navrhnu možné směry dalšího řešení mé práce. Návrhy dalšího řešení směřují k možnému rozšiřování komponenty o další funkce, které pomocí skriptovacího jazyka VBScript pracují se vzorky signálu.

8.1 Zhodnocení dosažených výsledků

V úvodu diplomového projektu jsem se nejprve seznámil a popsal základy teorie překladačů, gramatiky jazyků a výpočty hodnot atributů.

V druhé kapitole jsem nastudoval a zpracoval principy konstrukce překladačů, kompilační a interpretační překladače a strukturu překladače.

Dále jsem popsal známé skriptovací jazyky a to M-file pro MATLAB a Visual Basic – Scripting Edition. Pro zápis příkazů pro výpočet vzorců a celých algoritmů obsahujících podmíněné příkazy a cykly jsem použil ActiveX ovládací prvek Microsoft Script Control, který jsem rozšířil o vlastní funkce pro zpracování vzorků signálů. Následně jsem popsal tuto komponentu a její metody, vlastnosti a použití.

V šesté kapitole jsem navrhl a popsal software pro výpočet skriptu. Navrhl jsem aplikaci, do které byl přidán MSC poskytující skriptovací prostředí a popsal použité metody a vlastnosti. Tento prvek jsem rozšířil o vlastní funkce, které zpracovávají naměřené vzorky signálu. Dále jsem vytvořil novou ActiveX komponentu rozšířenou o nové funkce a také rozhraní potřebné pro přenos dat a správnou komunikaci s reálnou měřicí aplikací.

V předposlední kapitole jsem měl za úkol demonstrovat navržený software v reálné aplikaci a tak ukázat praktické a účelné využití své komponenty. Na ukázku jsem uvedl příklad použití skriptu v měřicí aplikaci.

V poslední kapitole jsem zhodnotil dosažené výsledky a navrhl směry dalšího řešení.

8.2 Návrh směrů dalšího řešení

Směry dalšího řešení vedou k vytváření nových funkcí a tím tak k neustálému rozšiřování mé práce. Výhoda této komponenty spočívá v tom, že je kompatibilně použitelná v různých reálných aplikacích a její rozšíření je velice snadné i pro člověka, který tuto komponentu nezná.

9 Závěr

Hlavním cílem této práce bylo vytvořit skriptovací jazyk, který dokáže matematicky zpracovávat naměřené vzorky signálu v reálné aplikaci.

V úvodu mé práce jsem se nejprve seznámil s teorií překladačů a principy konstrukce překladačů potřebné pro pochopení problematiky zadané práce. Tuto teorii jsem nastudoval a popsal v druhé a třetí kapitole.

Úkolem následující kapitoly bylo popsat známé skriptovací jazyky, např. M-file pro MATLAB a Visual Basic – Scripting Edition (VBScript). Na základě těchto vzorů jsem navrhl svůj skriptovací jazyk, který umí zápis příkazů pro výpočet vzorců nebo celých algoritmů obsahujících i podmíněné příkazy a cykly. Bylo tedy více možných řešení, které jsem musel zvážit. Rozhodnul jsem se pro variantu použít hotový ActiveX prvek Microsoft Script Control (MSC), který jsem rozšířil o potřebné vlastní nové funkce pro zpracování vzorků signálů.

Zabýval jsem se podrobně prvkem MSC, popisem jeho metod, vlastností a způsobu použití v programu. Tento ActiveX prvek poskytuje skriptovací prostředí pro uživatelské aplikace v jazyce VBScript. Tento jazyk nabízí kromě výhod práce se signály, také výhody práce se systémem Windows (kopírování souborů, automatické zálohování, vymazávání, instalace), automatizovaně spravuje web a databáze, lze si ulehčit práci při správě sítě atd. Já jsem se při tvorbě komponenty zaměřil na matematické funkce.

V následující kapitole jsem měl za úkol navrhnout aplikaci pro výpočet skriptu užitím objektového programování. Tato aplikace byla v jednoduchosti navržena tak, aby se dala později v reálné aplikaci používat bez pochopení vnitřních funkcí. Byly zde použity metody a vlastnosti prvku MSC a také objektové třídy, ve kterých jsou naprogramované nové funkce pro práci s vektory a maticemi. Původní prvek MSC s doplňky byly zkompilovány do ActiveX komponenty, která se tak může využívat v různých reálných aplikacích.

Předposlední úloha zadání diplomové práce požadovala v začlenění této komponenty do reálné aplikace tak, aby demonstrovala vlastnosti vytvořeného skriptovacího jazyka. Na ukázku jsem uvedl příklad využití skriptování v měřicí aplikaci.

V závěrečné kapitole jsem zhodnotil dosažené výsledky a navrhl směry dalších řešení.

Použitá literatura

AHO, V. A., SETHI R., ULLMAN D. J.: *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986, 796 s., ISBN 0-201-10088-6.

BABIUCH, M.: *Interpret Basicu s možností krokování*, Diplomová práce, Ostrava: VŠB 1998.

FARANA, R., SMUTNÝ, L., VÍTEČEK, A., VÍTEČKOVÁ, M.: *Zpracování závěrečných prací z oblasti automatizace a informatiky*, Ostrava: 2004, 114 s., ISBN 80-248-0557-X.

HABIBALLA, H.: *Překladače*, Studijní materiály pro distanční kurz, Ostrava: OU 2005.

KOCICH, P., GÜRTLER, M.: *1001 tipů a triků pro Visual Basic*, Praha: Compiuter Press, 2000, 328 s., ISBN 80-7226-368-4.

MELICHAR, B.: *Tvorba překladačů*, skriptum Praha: ČVUT 1996.

MELICHAR, B.: *Konstrukce překladačů*, skriptum Praha: ČVUT 1996.

MORKES, D.: *Visual Basic 6.0: Pro střední školy*, Praha: Compiuter Press, 2000, 164 s., ISBN 80-7226-312-9.

PETROUTSOS, E.: *Visual Basic 6: Průvodce programátora*, Praha: GRADA Publishing, 1999, 479 s., ISBN 80-7169-801-6.

PETROUTSOS, E.: *Visual Basic 6: Průvodce zkušeného programátora*, Praha: GRADA Publishing, 1999, 633 s., ISBN 80-7169-802-4.

POKORNÝ, J., KVOCH M.: *Programování ve Visual Basic 6.0*, České Budějovice: Koop, 2000, 373 s., ISBN 80-7232-044-0.

RŮŽIČKA, M.: *Návrh algoritmu pro sémantické akce při výstavbě interpretu metodou rekurzivního sestupu*, Diplomová práce, Brno: MZLU 2006.

TŮMA, J.: *Signal Analyser, the software support for education of signal processing*, Acta Montanistica Slovaca, 2003, vol. 8, no. 4, s. 159-161.

TŮMA, J.: *Nápověda k programu Signal Analyser*, dokumentace Ostrava: VŠB-TU 2002.